A Secure Isolation of Software Activities in Tiny Scale Systems

Von der Fakultät 1 MINT - Mathematik, Informatik, Physik, Elektro- und Informationstechnik
der Brandenburgischen Technischen Universität Cottbus - Senftenberg

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
(Dr. rer. nat.)

genehmigte Dissertation

vorgelegt von

Dipl.-Informatiker

Oliver Stecklina

geboren am 22.09.1976 in Cottbus

Gutachter: Prof. Dr. rer. nat. Peter Langendörfer

Gutachter: Prof. Dr.-Ing. habil. Jochen H. Schiller

Gutachter: Asst. Prof. Ph.D. Aurelien Francillon

Tag der mündlichen Prüfung: 28.10.2016

Abstract

With the introduction of the Internet at the end of the last century the modern society was fundamentally changed. Computer systems became an element of nearly all parts of our daily live. Due to the interconnection of these systems local borders are mostly vanished, so that information is accessible and exchangeable anywhere and at anytime. But this increased connectivity causes that physical fences are not longer an adequate protection for computer systems. Whereas the security of commodity computer systems was improved continuously and similarly with their increased connectivity, deeply embedded systems were then and now mostly protected by physical fences. But the ubiquitous availability of embedded systems in personal and commercial environments makes these systems likewise accessible and moves them strongly into the focus of security investigations. Deeply embedded systems are usually equipped with tiny scale micro controllers, which are limited in their available resources and do not feature secure mechanisms to isolate system resources. Hence, a single error in a local software component is not limited to the component itself, instead the complete system may be influenced. The lack of resource isolation makes tiny scale systems prone for accidental errors but in particular vulnerable for a broad variety of malicious software. For a safe and secure operation of computer systems it is strongly recommended that software components are isolated in such a manner that they have access only to those resources, which are assigned to them. Even though a substantial number of approaches in the context of embedded system's safety were investigated during the last fifteen years, security was mostly neglected. This thesis is focused on security aspects where malicious software wittingly tries to bypass available protection mechanisms. The thesis introduces a security platform for tiny scale systems that enforces an isolation of software components considering security aspects. Due to the limited resources of tiny scale systems the proposed solution is based on a co-design process that takes the static and predefined nature of deeply embedded systems into account and includes hardware, compile-time, and run-time partitions to reduce the number of additional run-time components, to avoid performance drawbacks, and to minimize the memory as well as the components footprint overhead. To prove the applicability of the presented platform it was applied and evaluated with two real applications. In addition, an investigation of technologies of commodity computer systems that are suitable to build secure systems is presented. The thesis analyzes their enforcement based on the features provided by the introduced security platform. The contributions of this thesis include an enforcement of a security isolation of system resources on tiny scale systems and enable the development of a broad variety of secure tiny scale system applications.

Zusammenfassung

Mit der Einführung des Internets am Ende des letzten Jahrhunderts hat sich in der heutigen Gesellschaft ein nachhaltiger Wandel vollzogen. Computer-Systeme wurden Bestandteil in nahezu allen Bereichen unseres täglichen Lebens. Durch die zunehmende Vernetzung der Systeme sind räumliche Grenzen weitgehend verschwunden, so dass die Informationen überall und jederzeit verfügbar sind bzw. verändert werden können. Diese erhöhte Konnektivität bedingt jedoch, dass der Schutz der Computer-Systeme durch physische Maßnahmen nicht mehr gewährleistet werden kann. Während die Sicherheit von alltäglichen Computer-Systemen kontinuierlich und in nahezu gleicher Weise zu ihrer gestiegenden Konnektivität verbessert wurde, sind tief eingebettete Systeme damals wie heute meist durch physikalische Maßnahmen geschützt. Die allgegenwärtige Verfügbarkeit von eingebetteten Systemen in persönlichen als auch in kommerziellen Umgebungen macht diese Systeme jedoch in gleicher Weise für jedermann zugänglich und macht sie damit zu einem zentralen Bestandteil der aktuellen Sicherheitsforschung. Tief eingebettete Systeme sind in der Regel mit kleinen Mikrocontrollern ausgestattet, die über begrenzte Ressourcen verfügen und keine sicheren Mechanismen zur Trennung von Systemressourcen bereitstellen. Hierdurch ist ein einzelner Fehler in einer lokalen Softwarekomponente nicht auf diese Komponente beschränkt, sondern beeinträchtigt nicht selten das gesamte System. Diese Schwäche macht eingebettete Systeme anfällig für zufällige Fehler, aber auch insbesondere anfällig für eine Vielzahl von bösartiger Software. Für einen stabilen und sicheren Betrieb von Computer-Systemen ist es zwingend erforderlich, dass Software-Komponenten in einer Art und Weise isoliert werden, dass sie nur Zugriff auf jene Ressourcen haben, die ihnen zugeordnet wurden. Wenngleich in den letzten fünfzehn Jahren eine beträchtliche Anzahl von Ansätzen zur Erhöhung der funktionalen Sicherheit in tief eingebetteten Systemen untersucht wurden, wurde die Sicherheit gegenüber Angriffen Dritter weitestgehend vernachlässigt. Die hier vorliegende Arbeit konzentriert sich auf Sicherheitsaspekte zur Abwehr von bösartiger Software, die wissentlich versucht verfügbare Schutzmechanismen zu umgehen. Die Arbeit stellt eine Sicherheitsplattform für kleine tief eingebettete Systeme bereit, die eine Trennung von Softwarekomponenten unter Berücksichtigung von Sicherheitsaspekten erzwingt. Aufgrund der begrenzten Ressourcen dieser Systeme basiert die vorgeschlagene Lösung auf einem Co-Design-Prozess, der den statischen und vordefinierten Charakter von tief eingebetteten Systemen berücksichtigt. Der Prozess beinhaltet Hardware-, Compile-Zeit- und Laufzeit-Komponenten um die Zahl der zur Laufzeit notwendigen Komponenten möglichst gering zu halten. Hierdurch sollen nachteilige Einflüsse auf die Laufzeit, den Speicherplatzbedarf sowie die Größe von Hardware-Komponenten minimiert werden. Um die Verwendbarkeit der präsentierten Plattform nachzuweisen, wurden zwei reale Anwendungen auf diese portiert. Zusätzlich

wurden etablierte Technologien zum Bau von sicheren Systemen hinsichtlich ihrer Verwendbarkeit auf tief eingebetteten Systemen analysiert. Hierzu wurde deren Umsetzbarkeit unter Nutzung der durch die Sicherheitsplattform bereitgestellten Sicherheitsfunktionen untersucht. Der wesentliche Beitrag dieser Arbeit beinhaltet die Bereitstellung einer sicheren Trennung von Systemressourcen auf kleinen, tief eingebetteten Systemen, so dass eine Entwicklung einer Vielzahl von sicheren Systemanwendungen auf diesen Systemen möglich wird.

Table of Contents

Acknowledgments

My years at the IHP in the system design group as a research assistant were some of my most inspiring and instructive years. The result of those years - my thesis - would not have been possible without the support of my professor and my great colleagues, to whom I want to express my special thanks.

My professor *Peter Langendörfer* enables and supports me and my colleagues in pursuing own research interests. It was the freedom and self-responsibility he granted to us that made us such a cracking good group of researchers, I have enjoyed to work in. Together with my colleagues *Nicole Todtenberg*, *Thomas Basmer*, *Dieter Genschow*, *Stephan Kornemann*, and *Frank Vater* we built an ecosystem that includes applications, an operating system, a sensor node platform, as well as a tiny scale soft-core processor. These components were parts of my daily work at the IHP and formed the basement of my security platform for deeply embedded systems. A special thanks to *Dr. Michael Methfessel*, he was a great colleague in a research project and an very instructive supporter in some of my publications.

Special thanks to the students *Hannes Menzel*, *Erik Bergmann*, and *Andreas Krumholz*. They had to suffer my supervision of their master theses. However, they were all the time very good counterparts for inspiring discussions and helped me to devise basic concepts of my research. A further thanks to *Kai Lehniger* who helped me to made the tinyVLIW8 processor more usable by implementing CoMeT transformation modules.

My wife *Katja* supported and motivated me in the most challenging phases of my thesis. She spent a lot time for proof-reading and discussed with me the content of my thesis. Due to her wide and considerable knowledge within the area of safety she was always an inspiring counterpart in technical discussions.

Cottbus, May 2016

CHAPTER 1

Introduction

The ubiquitous availability of information technology (IT) in our daily life has changed our modern society in a significant manner. The Internet and mobile information systems make information accessible anywhere and at any time. Whereas in the last century computer systems have moved from enterprise mainframes in data centers to personal computers (PCs) at home, in the 21st century systems become smaller and ubiquitous. In the last recent years the terms *internet of things (IoT)* and *cyber-physical systems (CPSs)* were minted. The terms carry the proceeding penetration of computer systems into our daily life in their names. They reflect the current most innovative trend of modern system engineering. Embedded micro controller units (MCUs) are the core component of the IoT and CPSs. With eight billion units deployed in the year 2000, MCUs pose the lion's share of micro processors as opposed to 150 million general-purpose computers sold in the same year [Ten00].

The title of this thesis is built by using two terms: tiny scale system (TSS) and software activity (SA). Both terms are not standardized and may be used with different meanings in different documents. Therefore, a short description of the meaning of these two terms are given at the very beginning of this thesis.

> **Tiny scale systems (TSS)** are systems, which are very restricted in their available hardware resources, execute small-scale software without multi-user support and process a tiny volume of data. The major application areas of TSSs are tightly coupled with embedded systems.
>
> **A software activity (SA)** is a software component that is a scheduled computation [CCJ+07], or an event-driven function [LMP+04]. It consists of a single or a multitude of functions that are combined to perform a specific task in a TSS.

The applications of TSSs are usually invisible for common users and communicate with other computer systems. Their application covers a broad variety from healthcare, aerospace, automotive, infrastructures, energy manufacturing, transportation, chemical processes, entertainment, and consumer appliances. They include, but are not limited to, monitoring and control systems. They are often deeply embedded in larger computer systems, are responsible for different kind of data or signal processing like encoding, encryption, or filtering and operate in most scenarios without any type of user interface. In case of complex applications a various number of autonomous systems are linked into a network. In contrast to common computer system, which are in a physically controlled and carefully administrated environment, TSSs are typically physically accessible and operate mostly unmaintained.

Since the ubiquitous penetration of TSSs an enforcement of a secure operation is a key-factor for the design and implementation of these systems. This thesis introduces a secure isolation of SAs to enforce a secure and safe operation of TSSs in their daily applications.

1.1 The ubiquitous of cyber-physical systems (CPSs)

The introduction of embedded systems is tightly coupled with the third industrial revolution. The revolution has been started in the early seventies of the last century and has brought programmable logic controllers and computer networks to the industrial plants. The core of the third revolution was the process automation, the control of the physical world. The fourth industrial revolution had started at the beginning of the second decade of the 21th century. It is claimed that fundamental innovation of the fourth revolution was the introduction of CPSs, the connection of the physical world with the virtual one.

In the following we will give a brief description to the differences between traditional embedded systems and CPSs. A traditional embedded system gets a well-defined measuring and control task. The system works dependably as a *black box* in harsh environments. It is equipped with an MCU with limited resources and various input/output interfaces. The R&D of embedded systems was focused on hardware interfaces, device drivers, multi-tasking, memory management, code optimization, and real-time OSs. In the last recent years the tasks of embedded systems have been enlarged. The systems were decentralized, got more autonomous intelligence and even became connected to the internet. For a differentiation of this new class of devices the California University of Berkeley has formed the term CPS [Sch14]. In comparison to traditional embedded systems, the number of physical components involved in a CPS is higher. These systems are focused on the link between the computational and the physical elements. Their innovation is driven by three core technologies: computation, communication, and control. It can be summarized that a system can be called a CPS if it fulfills the three 'C', shown in Figure 1.1.

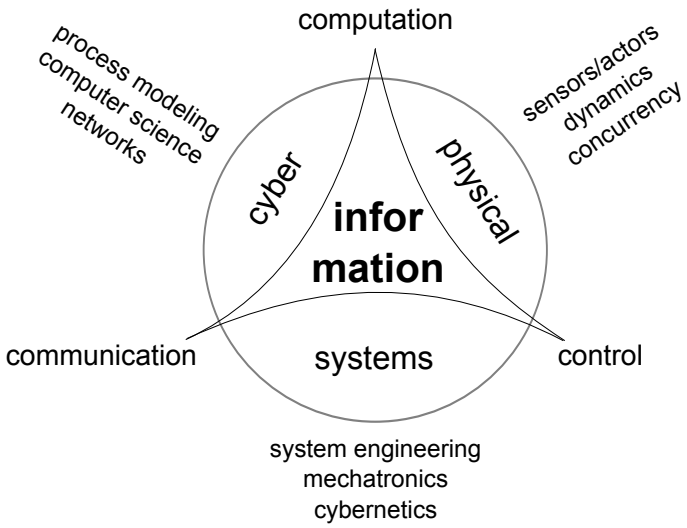

Fig. 1.1: Core technologies of cyber-physical systems [Sch14]

The typical structure of a deployed CPS is sketched in Figure 1.2. The example features three main parts corresponding to the three 'Cs': a physical plant, one or more computational platforms, and the network fabric. The system has two networked platforms, each with its own sensors or actuators. The action taken by the actuators affects data captured by sensors through the physical plant. In the example, platform 1 is controlled by platform 2 and the other way around, so that the system forms a feedback control loop through physical and virtual networks [LS14].

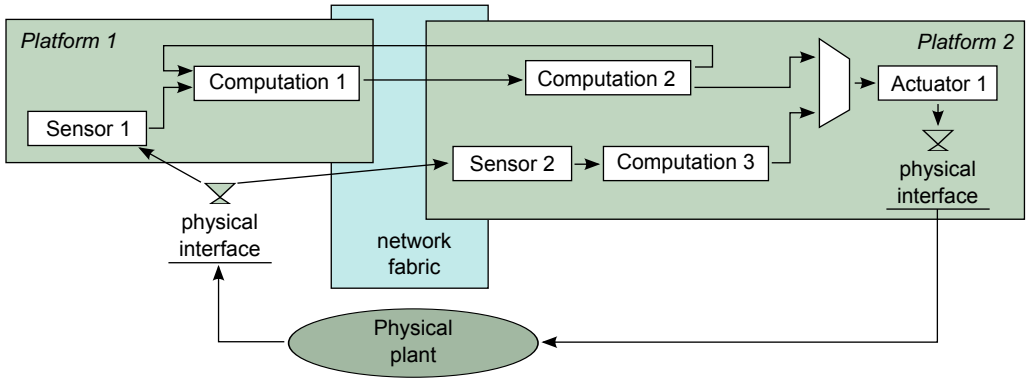

Fig. 1.2: Example structure of a cyber-physical system [LS14]

However, as the interaction between the physical and cyber world increases, physical systems become increasingly more susceptible to the security vulnerabilities in the cyber world [WYX+10]. The industry of computer security has started to concern about the security of CPSs. But these solutions are merely an adaptation of common systems from the cyber world. The digitalization of sensors and actuators as a TSS offers an additional window of vulnerability. As these TSSs are physically deployed outside buildings or without physical fences, they can be easily accessed. Furthermore, any digital system can be controlled from anywhere by using the technology of the cyber world. Although TSSs are limited in their resources a small network of systems is powerful enough to gather and process continuous data to assist systems. In fact, TSSs will always be vulnerable to do the bidding of attackers, to the detriment of their owners.

The threat caused by a TSS as part of a CPS may range from a privacy injury through an hijacked webcam up to a direct danger for a human life by manipulated automobiles or medical devices. A typical webcam is equipped with an MCU, which configures the imaging chip and transfers the captured data to the host device, e.g. a connected PC or a network controller. An hardware setup is shown by Figure 1.3. Beside the MCU and the imaging chip a flash memory may be used to store the firmware image of the MCU and the imaging chip as well. The system features a LED indicator light that signals activity to give the user a feeling of security and privacy. But the authors of [BC13] have shown that such a system can be manipulated. Although the LED is hard-wired with the imaging sensor, a remote attacker can manipulate an embedded controller to capture imaging data in a way that a user in front of it will not recognize it. In a private environment such an attack may be a personal nightmare [And13], in companies the same attack can be used to gather critical information, which can be an endangerment for the business.

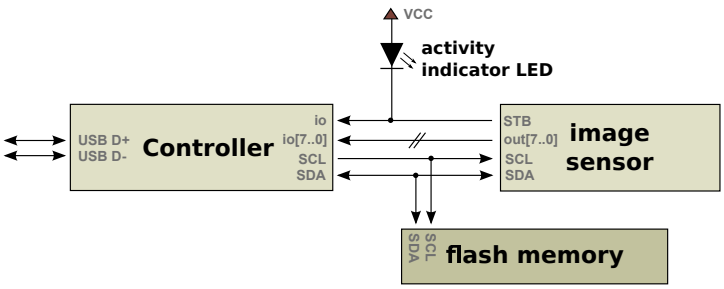

Fig. 1.3: Block diagram of a digital webcam. The activity indicator LED can be controlled by an adversary by hacking the embedded controller to cover harmful activities.

Whereas hacking a webcam may be an attack without risk of real life, modern computer systems are equipped with a large number of MCUs, which control peripheral components. A more risky attack on the battery controller of laptop systems is presented by Miller et. al [Mil11]. Because of the nature of Lithium-Ion batteries, changes to the controller's firmware may cause safety hazards such as overcharging, overheating, or even fire. Considering that these systems are also used in larger systems such as industrial plants or battery electric vehicles (BEVs), attacks on TSSs may threaten our everyday life. Especially modern automobiles are no longer mere mechanical devices. They include dozens of digital computers coordinated via internal networks. The authors of [KCR+10] presented a software tool called *CarShark*, which can be used to hijack a car remotely. They were able to disable brakes or to give false sensor readings to the control systems. They were able to threaten the real life of a driver or other persons. A more direct attack in the real life of humans was presented by Leavitt. Hackers were able to hack medical devices implanted in a human body. They have made use of the wireless communication to penetrate the system [Lea10].

With a look on traditional computer systems we get an idea how we can try to provide security on TSSs. Rutkowska [Rut08] categorizes attempts on traditional computer systems into three broad categories:

a) security by correctness,
b) security by isolation, or
c) security by obscurity.

The assumption of security by correctness is obvious. The only problem is that the implementation of a correct software is very hard and associated with strong restrictions for the used tools and languages. People, from the very beginning, have also taken an approach that is based on isolation. The idea is to split systems into small pieces and to make sure that each piece has access only to those resources that it owns. Some people propose a security by obscurity or randomness. But, it's all about making a system more unfriendly to an attacker [Rut08].

My belief is that a fundamental step to build secure CPSs is a fine-grained isolation of software components in TSSs. I'm convinced that a basic isolation technology can increase the system's security in a significant manner. Hence, the principal part of my thesis is focused on providing a secure and applicable isolation in TSSs.

1.2 Contribution of this thesis

The aim of this thesis is to design, implement and evaluate a secure platform for the domain of resource-restricted, deeply embedded devices. The addressed class of devices does not feature a hardware-based memory protection and shares all resources in a single address space as typically used in digital sensors of CPSs. The presented approach is based on a tailor-made MPU for soft-core processors and a software-based solution for commodity MCUs. Beside the MPU the major contributions of the proposed platform include the following objectives:

Fine-grained isolation The platform provides an isolation of memory resources that takes the needs of TSSs into account. It considers MCU's integrated peripheral units, which require a byte-granular isolation of resources. In addition and in contrast to other

approaches, the platform provides a secure isolation on write as well as on read access to any resources.

Capability-based control flow The control flow of TSSs follows the data that are needed to process the demanded task. Because of using an isolation of resources a capability-based protection scheme becomes feasible. It enforces control flow integrity on cross-isolation communication and further protection of private information.

Kernel-less system Cross-isolation communication includes a switch to foreign protection domains. A kernel is controlling these switches on commodity systems. But each switch causes a significant overhead that must be avoided on TSSs. Therefore, the proposed platform allows a direct switching and operates without a system kernel to minimize the cross-isolation communication overhead.

Compile and run-time co-design The enforcement of a secure isolation and a capability-based control flow are usually payed by a significant run-time overhead. But since applications of TSSs are mostly fully available in source a compile and run-time co-design process can be used to shift complex operations into the compilation step and add only a minimal run-time overhead instead.

1.3 Publications related to this work

The concept and results of this Ph.D. thesis and preliminary works have been published in nine different papers on national and international conferences. Two of these papers sketched the basic ideas of my thesis. The other papers are preliminary and surrounded work or present individual results. Beside the papers I supervised three Master's theses that are strongly related to the work of my Ph.D. thesis. In the following I will give a brief overview about the published papers and the supervised Master's theses and will explain their relation to my Ph.D. thesis.

The basic concept presented in this thesis was initially published in the poster session on the *Internal Conference on Pervasive and Embedded Computing and Communication Systems (PECCS)* in 2011 [SLM11]. It includes ideas of a secure separation of address spaces on resource-restricted devices. Within the following four years the ideas were developed to the final concept presented within the *Annual Ph.D. Forum on Pervasive Computing and Communications at the International Conference on Pervasive Computing and Communications (PerCom)* in 2015 [Ste15a].

The initial approach of a hardware-based MPU for an MSP430 MCU was developed in SystemC by Hannes Menzel. He presented the results of his work in his Master's thesis in 2010 [Men10]. The approach was evaluated in a hybrid simulation environment for an MSP430 (HSE430), which was developed by us in a preliminary work. The HSE430 combines a Java-based instruction set simulator (ISS) with the SystemC simulation environment. We will introduce the system in detail in Section 7.2.2.1. The work was presented in a poster session on the *International Conference on Design and Diagnostics of Electronic Circuits and Systems (DDECS)* in 2011 [SMV+11]. The design of a tailor-made memory protection unit

for low power MCUs was presented on the *International Symposium on Industrial Embedded Systems (SIES)* in 2013 [SLM13]. The results of the Master's thesis of Hannes Menzel were taken by Erik Bergmann to provide a framework for an isolation of software activities on these class of devices. The Master's thesis presents the results of this work [Ber12].

During the IQlevel projects we developed the "Lego-like" sensor node for an innovative high quality level meter [IHP10]. The sensor node has a modular design, which can be easily adapted the different application needs and environments. It features an field programmable gate array (FPGA) devices, which was used to evaluate our initial hardware implementation. The FPGA module replaces the MCU and features the same interface so that an application can be compared to its original implementation. The FPGA module was developed during the Aeternitas projects [IHP12] and the sensor node was presented on the *International Conference on Sensor Networks* in 2012 [SGG12].

Beside the configurable sensor node we started the development of a configurable, low power sensor node operating system within the IQlevel project. The work was presented at the *GI/ITG KuVS Fachgespräch* in 2014 [SKK14]. In 2015 langOS was released on the sourceforce online repository [Ste15b]. An extended configuration scheme and compilation model for langOS was developed by Andreas Krumholz within his Master's thesis in 2015 [Kru15]. I used langOS and its extensions during various projects during the time of my Ph.D. thesis. Therefore, langOS is also used as the basic OS of the example applications presented within this work. We will briefly introduce langOS in Section 6.1.1.

The Aeternitas project aimed to provide a secure, ultra low power wake-up radio for embedded devices. During the project we developed a secure wake-up scheme based on the time-based one-time password (TOTP) algorithm. The work was presented on the *International Workshop on Mobile Systems and Sensor Networks for Collaboration* in 2014 [SKM14]. I used this application within my Ph.D. thesis as an example of a secured TSS. The SWUR approach was initially implemented as a pure software solution for an MSP430 device. Later a hardware/software co-design based on a soft-core processor tailor-made for deeply embedded controlling tasks was developed. The soft-core processor was also developed within the project and evaluated on the FPGA module on the configurable sensor node. The work was presented on the *Euromicro Conference on Digital System Design (DSD)* in 2014 [SM14]. I used the soft-core processor to illustrate the assembling of our MPU.

Section 7.2.2 of my thesis provides a system evaluation based on the HSE430 and on the soft-core processor. A very similar evaluation was done by me for an implementation of an intrinsic code attestation for embedded devices. The work was presented on the *International ICST Conference on Security and Privacy in Communication Networks (SecureComm)* in 2015 [SLV+15].

1.4 Structure of the thesis

The remainder of this thesis is structured in seven chapters:

Chapter 2 presents the scope of this work. The first section discusses security threats and weaknesses of TSSs. It is followed by a brief introduction of basic technologies for building secure systems and a selection of applicable technologies for TSSs. The chapter gives also

an overview about the class of micro controllers, which are addressed by this thesis. The chapter closes with a presentation of two small example applications that are used during the thesis to illustrate the proposed concepts.

Chapter 3 discusses the state of the art related to this thesis. It starts with an overview about access control in computer security. As a primary goal of this thesis, the restricted access to resources of TSSs commodity mechanisms are analyzed. The analysis includes hardware-based memory management as well as memory protection technologies, software technologies as safe languages, static code analysis, fault isolation, sandboxing, and virtualization. The chapter is finished with a short overview about operating systems of sensor nodes and security technologies of TSSs.

Chapter 4 introduces the platform for security enhanced TSSs. The presented platform is based on data spaces, as general container of system resources, the security nucleus, as the central gate keeping component, the definition of a role-based security policy book, and a tailor-made build system to enforce security policies and a compile-time/run-time co-design. The chapter will explain this four basic principles in detail.

Chapter 5 describes the assembling of the security nucleus on real systems. We divide the security nucleus in a software-based nucleus gate and a memory protection nucleus. The chapter starts with an introduction of the MSP430 and tinyVLIW8 processor cores. Both cores are target platforms of the hardware-specific memory protection nucleus. In the following, the assembling of the nucleus gate and the memory protection nucleus is presented.

Chapter 6 states the port of the two example applications of Chapter 2 on the security presented presented in the chapters before. The chapter includes a description of the enforcement of a fine-grained access control model to these two real examples. Since the first example application has been implemented on the langOS operating system, the chapter gives a brief introduction to langOS first.

Chapter 7 gives an evaluation of the proposed platform. In the first part of this chapter, we give a qualitative, high-level security evaluation of the presented approaches. In the following, we will discuss the platform costs in physical resources and performance. We introduce the HSE430, which is used to evaluate performance benchmarks on the MSP430 MCU.

Chapter 8 concludes this thesis and summarizes their primary goals and the achieved benefits. Furthermore, the chapter gives an outlook on further work, which could not be included in this Ph.D. thesis.

CHAPTER 2

Goals and assumptions

Before we start with a detailed description of common technologies related to the topic of this thesis, we want to focus our goals and assumptions. Hence, this chapter starts with an overview about security threats and weaknesses of computer systems and TSSs. It is followed by a section explaining technologies needed to build secure platforms on common computer systems. On these platforms most malicious effects of security threats and weaknesses can be entirely avoided or can be reduced to a minimum so that their occurrence can be neglected. But the use of these technologies on TSSs is still a big challenge.

For a better understanding about our notion of *tiny scale systems*, addressed by this thesis, this chapter includes a description of system assumptions and a threat model. The chapter is concluded by a brief introduction of two example applications. These applications are used to illustrate weaknesses of current systems as well as benefits of the presented approach.

2.1 Security threats and system weaknesses

When a TSS becomes part of a critical infrastructure (CI) or a CPS it will be subjected automatically to several types of attacks. Especially as part of a wireless sensor network (WSN), a TSS gets an exposed position to remote attacks. Any attacker has access to a broad variety of malicious mechanisms and technologies such as eavesdropping and modification of remote communication, installation of malicious application code or exploiting weaknesses to gain access to higher privileges. Although not all weaknesses are exploitable, some of the mechanisms can affect the reliability, dependability and safety of a TSS and the surrounding system in a significant manner.

This section gives a brief overview about security threats and system weaknesses. Although it is widely believed that most of these threats and weaknesses are difficult to use on TSSs, recent work has demonstrated that it is not impossible [FC08]. The attacks are grouped in local, remote, and tamper attacks. Local attacks assume a physical or remote access to the target that has weaknesses in hardware or runs vulnerable software. Remote attacks include all types of attacks on the network layer to confuse or cut communication connections. Tamper attacks include physical attacks on the system's hardware to gather stored data or engineering information.

2.1.1 Local attacks, software vulnerabilities

Local attacks exploit weaknesses in hardware or software to infiltrate a system or to gain additional privileges. Weaknesses in hardware are possible but very rare. Since TSSs, focused by this thesis, do not feature any hardware-based protection scheme hardware weaknesses

can be neglected. Hence, in the following local attacks based on software weaknesses are discussed.

There are two common ways to arrange malicious code within the victim's address space: inject it or miss-use existing code. In case of injecting malicious program code an attacker may use an unprotected buffer to impose a string with native malicious CPU instructions. In case of miss-use, an attacker can manipulate the program flow in that way that it executes existing program code to the victim's detriment [BBD06]. In both cases an attacker needs a way to redirect the normal program execution to its malicious code.

2.1.1.1 Stack smashing

In modern computers with the need of high-level programming languages the most important technique for structuring programs is a procedure or function. A procedure includes a portion of program code, a subprogram, that can be used in different program sections. A procedure call that invokes the subprogram, alters the program flow just as a jump does. But when finishing the subprogram the program flow returns to the instruction followed the call instruction. This high-level subprogram call is mostly implemented with the usage of a stack. In computer science, a stack is an abstract data type that has the property that the last object stored on the stack, `push`-operation, is the first object removed, `pop`-operation. The stack is also used for local variables, to pass function parameters, and to store the return value.

The most common form of security vulnerability on C implementations is to corrupt the execution stack by writing beyond the end of a local array. Code that does it is said to smash the stack and can cause a return from the routine to any address [One96]. Figure 2.1 illustrates the layout of the program stack before and after its smashing. The stack usually grows down from high to low addresses. The lower end of the stack is stored in the stack pointer, usually a processor register. On each `push`-operation the pushed value is written onto the current stack position and the stack pointer is decremented. In case of a subroutine call the current *instruction pointer* is pushed onto the stack, it is the program address where the program goes on after the subroutine call is finished.

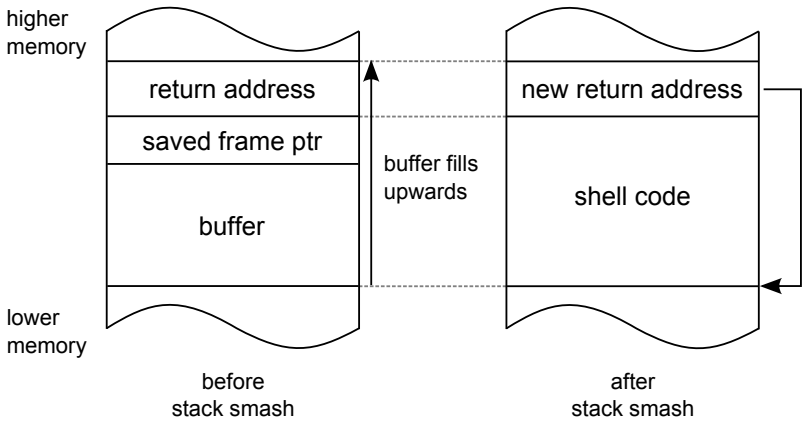

Fig. 2.1: Stack smashing attack [Ayc06].

Especially when the local buffer can be written by an input string, the attacker can insert a new return address and the program will jump to the new address instead of returning to its original address. Depending on the system's architecture the stack can be used to insert

the new program code as well. The attacker will set the return address to the program code inserted in the same buffer.

2.1.1.2 Function pointers

Similar to stack smashing, function pointers can be used to manipulate the program flow. A function pointer can be allocated anywhere and the attacker needs only to identify a vulnerable buffer in the near of the function pointer. He can use the buffer to manipulate the function pointer in a way similar to a buffer overflow. Later when the program makes a jump through the manipulated function pointer, it will jump to the location desired by the attacker.

2.1.1.3 Return-oriented programming (ROP) attacks

In the simplest and most common form of local attacks the injected program code and the activation record are combined in a single string. Early iterations of defense modified the memory layout of a program to make the stack non-executable. But this can be bypassed easily by using two separated buffers. The first buffer is filled with the injected program code and the second includes the activation record only. However, writing a local buffer requires that the buffer is located on the heap, in the BSS section[1], or on the stack. Depending on the system's architecture, a code injection might be difficult or impossible. So systems with physically separated data and code memories, e.g. Harvard architecture, systems with a non-executable data section or with a memory protection mechanism as the no-execute bit (NX-bit) or Write-Xor-Execute will prevent simple code injection attacks.

A technique to bypass these protection mechanisms is the return-oriented programming (ROP) attack [Des97, Sha07]. Instead of injecting program code, existing code sections are used to build the desired program sequence. Especially on Unix-based systems, functions of the standard C library, *libc*, could be used. An attack based on libc functions is also known as a *return-to-libc* attack. The library is loaded in nearly every system and contains a broad variety of suitable functions. In principle any available code, either from programs or from libraries, could be used. With carefully arranged values on the stack, an attacker can cause a series of functions to be invoked [Ner01]. The return-oriented programming, as illustrated in Figure 2.2, uses snippets of code, called *gatgets*, located at the end of functions to generate building blocks to assemble an arbitrary code sequence.

TSSs with their single address space feature a large amount of code usable in ROP attacks. Francillon et al. presented that this technique is also applicable on TSS with a Harvard architecture [FC08]. Furthermore, especially on MCUs, the code of a bootloader can be used to build gatgets on TSSs similar to return-to-libc attacks. Due to the fact that the same bootloader is used on MCUs of the same model and version, it is easy to purchase a chip and analyze its bootloader [GF09].

2.1.2 Non-local attacks

The untrusted environment of the applications of TSSs makes these systems vulnerable to non-local attacks. These attacks may be primarily directed against the communication over

[1] The BSS segment is part of the data section and contains statically-allocated variables.

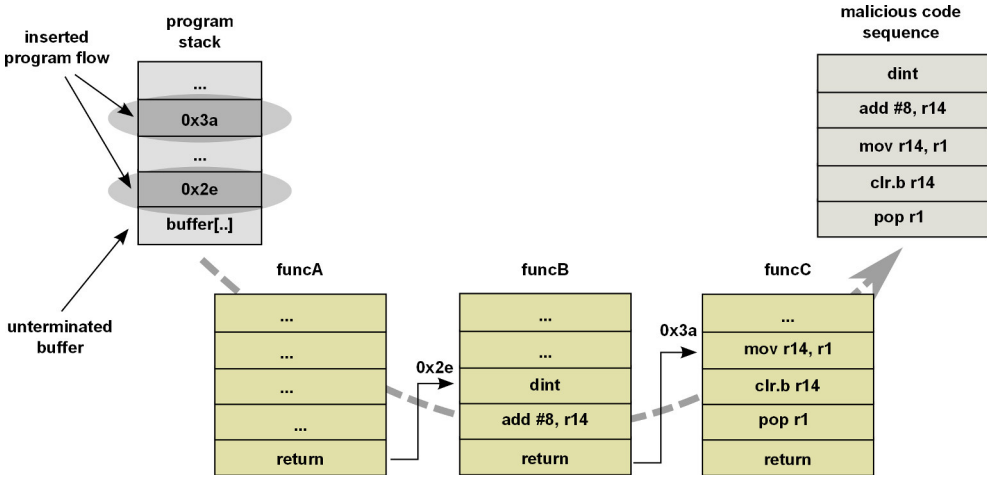

Fig. 2.2: In return-oriented programming (ROP) attacks, an attacker gains control of the call stack to hijack program control flow.

an open and unprotected channel or against physical protection mechanisms. Under these circumstances a secure isolation of software activities will not be able to hold an adversary off. Therefore, these types of attacks are mainly out of the scope of this thesis. Nevertheless, the presented approach may help to make an attack more difficult.

2.1.2.1 Remote attacks

Because of the embedded nature of TSSs, their interface to the outer world is mostly limited to machine-to-machine communication. In particular, WSNs use a shared and publicly accessible medium. Therefore, any data received over these interfaces must be handled with special care. In the last recent years several attacks became known. Especially WSNs are vulnerable to several attacks. Embedded systems such as vehicle controllers are also vulnerable, if internal communication links are used.

In most WSNs an attacker can easily inject malicious packets and impersonate another sender, which is referred to a spoofing attack. Furthermore, an attacker can easily eavesdrop on communication, record packets, and replay the potential altered packets [HPJ03]. Embedded devices destined for an unmaintained operation in harsh environments should already be designed to continue functioning in the presence of faults. This robustness against physical challenges may prevent some classes of denial of service (DoS) attacks, which refers to an adversary's attempt to disrupt, subvert, or destroy communication [WS02].

However, there are various preventive approaches that have been applied to protect WSNs [PSW+01, HPJ03, KSW04, YZC08]. Moreover, there are no guarantees that the preventive methods are able to hold an adversary off. Hence, mechanisms of safe and secure software architectures are demanded to limit the possible impact.

2.1.2.2 Tampering attacks

In case of having physical access to a device, attacks on embedded microcontrollers are almost trivial. For example, the bits including the memory lock bit can be erased by focusing

UV light on it [AK96]. Security critical devices as smartcards are slightly harder to attack. But in the last years several different techniques for attacking physical devices have been successfully approved. In general tamper attacks on physical devices can be categorized into non-invasive, semi-invasive and full-invasive. The non-invasive attacks such as power analysis do not require a sample preparation. Resilient software implementations can limit the impact of such attacks. In case of a semi-invasive or a fully-invasive attack, the attackers isolate vulnerable areas of the integrated circuit (IC). The attacks are focused on a small fraction of the hardware implementation by directly targeting the internal logic that includes the demanded information. A number of less expensive techniques for attacking tamper resistant devices is also known [AK98]. Skorobogotav and Anderson describe in depth tampering attacks on microcontrollers [SA03]. Becher et al. evaluated different physical attacks against sensor node hardware [BBD06].

Since building a tamper-resistant device and using them effectively is much harder than it looks, approaches that provide a tamper-evident execution environment were introduced. Such an environment will protect the confidentiality of a program and its data as well as it will be able to detect tampering [SCG+03]. Although such a tamper-evident execution environment is provided by a microprocessor by using hardware extensions such as encryption, physical random functions or physically unclonable functions, it still makes extensively use of software functions. Thus software attacks on these systems are still a major critical issue and an enhanced protection scheme is particularly demanded.

2.2 The threat model

In general, TSSs addressed here are deployed in untrusted environments. Although it may be possible to design a system environment that guarantees integrity of each component that interacts with our TSS, we are convinced that such a system will be too restrictive and is not applicable to the majority of distributed, embedded systems. Due to the fact, that an adversary will always gain access to a communication channel used by a TSS, we must assume that any communication with a TSS is untrustworthy. In wireless systems anyone can eavesdrop traffic, inject malicious messages, and replay old messages. Hence, especially for wireless interfaces no trust assumptions can be placed on the incoming data [PST+02].

To provide security on a system with malicious software, we implant a security nucleus that decouples the underlying hardware from the software. The security nucleus may be based on an additional software component or an MCU extension. We do not address any kind of tamper attack. Any kind of attack in which an adversary has direct access to system's memory by using electrical probes or programming interfaces is out of the scope of our work. We presume that we are able to control any memory access, which is performed by an SA on the same TSS.

We assume that the system has a secure and static boot-strap, which initializes the system in a secure manner. It is mandatory that the initialization process of the trustworthy core components is finished before insecure components are initialized or started. Furthermore, it is important that the processing of unpredictable data has to be delayed as well. Unpredictable data include the whole network traffic, all digital sensor data, and any loadable program code segment. It is important that the initialization of the trustworthy components is done without the need of any unpredictable data. An adversary that has access to the communication

channel or the data source can manipulate the data in a way that a secure system initialization cannot be guaranteed.

2.3 Building secure systems

Security threats and system weaknesses are critical in all types of computer systems. Within the context of commodity server, desktop, and in particular mobile computer systems Härtig [Här02] argued that the technologies needed to build modern, secure systems must include small interface technologies, access-control contracts, tunneling, secure booting, effcient resource control, and virtual machines.

Small interface approach can be built by using two alternative approaches: μ-kernels and extensible systems. The μ-kernel approach separates the system in small pieces, whereas extensible systems use safe languages [BSP+95] or transaction-like mechanisms [SESS96].

Access-control based on contracts can be seen as a high level abstract descriptions of role-based access control (RBAC). In such a contract an object or a group of objects declare their needs and the specific functions that they provide.

Tunneling can be used as a technique to add a required property to a software component by using an additional layer. This may include an insecure communication channel that is used to transfer data. Hence, the provided security level of the software component that implements the additional layer can be ignored.

Secure booting ensures that a specific OS and a specific application is indeed running on a specific device by establishing a secure boot chain with a (hardware-based) trust anchor.

Effective resource control is progressed significantly in the real-time systems community. The technique can provide an effective defense against denial-of-service attacks as well. Especially in the area of TSSs an effective resource control becomes key.

Virtual machines provide a high level separation of software components by an emulation of a hardware architecture. However, the costs of emulating the hardware architecture are acceptable for powerful devices only.

It is important to separate mechanisms and policies in a proper way [WCC+74]. Mechanisms are a collection of functions and facilities that are necessary to enforce policies. This separation leaves the complex decisions and operations in the hands of the person who should make them such as the system designer. In a proper designed system, protection can be seen as a mechanism implemented in a device to ensure the integrity of an operation. This

protection includes the traditional read, write and execute capabilities but may include arbitrary capabilities as well.

The knowledge that includes the technologies to build a secure system is well-established in traditional OSs. We are convinced that in a crossover to TSSs these technologies are more or less applicable as well. The platform designed, implemented, and evaluated in this thesis, aims to provide equivalent technical components directly or indirectly. It is derived from three main properties of TSSs: event-driven applications, limited resources, and untrusted environments. Based on these properties and the techniques needed to build a secure system we propose five design principles:

- a tailor-made address space separation,
- program flow integrity,
- a minimal trustworthy component,
- fine-grained access control and
- an extended compilation module.

2.4 System assumptions

Before we start with an introduction of our secure platform and the corresponding related work, we will define the underlying system's architecture and the trust requirements. The goal of this work is to propose a general secure platform that is applicable to a tiny scale system (TSS). But the characteristics of TSSs cover a broad variety of devices and applications so that this work cannot address all of them.

In the last recent years mostly all designs of secure platforms had been focused on systems equipped with a minimal memory management unit (MMU) and commodity OSs derived from standard computer systems. We are convinced that a change to this powerful type of micro-processors will be not done in all critical systems. We motivate this by the broad use of low power micro-processor without an MPU and the conservative attitude of developers in the area of deeply embedded systems. Hence, a secure platform must be applicable on these systems as well. Therefore, the proposed secure platform of this thesis does not address any commodity OS running on micro-processors equipped with a proper MMU or at least MPU. We will illustrate that our approach is applicable on a common MCU.

2.4.1 Micro-processor architectures of tiny scale systems (TSSs)

We introduced the term TSS and explained that these systems are tightly coupled with deeply embedded systems. Their applications are primarily focused on controlling tasks and their data processing capabilities are quite limited. Common MCUs are usually multi-chip modules with integrated memories, reset and interrupt controllers, and peripherals. Nevertheless, the differences to micro-processors are fluently. We can see MCUs, which are based on common micro-processors, e.g. Intel 80186 (based on Intel 8086), XScale (ARM), or ColdFire (MC680xx), and MCUs, which are primarily developed for embedded systems, e.g. TI

MSP430, Atmel AVR, Infineon TriCore, XE166, or Intel 8051. These systems are cost, power, as well as latency optimized and based on 4-, 8-, 16- and 32-bit architectures. Especially in the area of low power applications 8- and 16-bit controllers are widely used.

All the micro-processors can be grouped in two fundamental architectures: the von-Neumann architecture and the Harvard architecture.

The von-Neumann architecture, also known as the *Princeton architecture*, was initially described in 1945 by John von Neumann [vN93]. It stores the program data as well as the instruction data in a single memory. The usage of a shared memory simplifies the system architecture but implies significant disadvantages. An instruction fetch and a data operation cannot occur at the same time and instruction can be overwritten by data operations.

The Harvard architecture is a computer architecture with physically separated instruction and data memories. In contrast to the von-Neumann architecture, the central processing unit (CPU) can read an instruction and perform a data operation at the same time, thus be faster for a given circuit complexity. In addition, the Harvard architecture provides two different address spaces for data and instruction memory. Hence, address X of the data memory is not equal to address X of the instruction memory. Therefore, the code memory can be manipulated by special instructions only.

The Harvard architecture is more suitable for embedded controller tasks and is used in most of the available MCUs, as Atmel AVR, Intel 8051, and Infineon TriCore. Nevertheless, the von-Neumann architecture is used as well, e.g. TI MSP430. Modern MCUs combine both architectures. The internal processor core is based on the more efficient Harvard architecture and the external interface implements the more simpler and cheaper von-Neunmann architecture.

During the design of an MPU for a secure isolation of software activitys (SAs) the used computer architecture of the processor core must be taken into account. Therefore, this thesis shall illustrate the integration of an MPU in an MSP430 with its von-Neumann architecture and in the tinyVLIW8 with a Harvard architecture.

Beside the computer architecture, the instruction execution/cycle must be considered. The instruction cycle is the process by which a CPU retrieves a program instruction, determines the action, and carries out those actions. In simple CPUs the instruction is executed sequentially: each instruction step is finished before the next one is started. In modern CPUs instruction cycles are rather executed concurrently in parallel by implementing an instruction pipeline as well as in a dynamic order by applying an out-of-order execution.

Instruction pipelining is a technique used in computer architectures to increase their instruction throughput. A basic instruction is broken in instruction steps, which are executed concurrently in a pipeline. This model starts an instruction before the last instruction was completed, which can result in the situation that the result of an instruction is needed before the instruction was finished. The problem is known as a hazard and can be solved by instruction delay, instruction reordering, or speculative execution. Instruction delays are unproblematic, but the other two workarounds can raise instruction sequences that may confuse a memory protection scheme.

Out-of-order execution is a technique used in high performance computers. Instead of the sequential instruction execution defined by the program text, the CPU executes instructions in an order governed by the availability of input data and functional units. It is used to avoid situations where the CPU must wait/idle to retrieve resources from a previous instruction [HP86].

All the technologies to improve the system's performance make memory protection more difficult. Without additional care, a memory access might not be assigned to the causing instruction. But because of the complexity in hardware and the unpredictable run-time behavior of these instruction cycles a use in MCUs of TSSs is very rare. In this thesis, we assume that an MCU used in TSS does not make use of these techniques, all instructions are completed before a new one is started, and the execution is ordered in the same way as defined in the program text.

2.4.2 Soft-core processor

In electronic design an intellectual property (IP) core is a reusable unit of logic, cell, or chip layout design. IP cores are building blocks used in an application specific integrated circuit (ASIC) or in an FPGA design. A soft-core processor is a synthesizeable IP block that features a processing core of a proprietary or commodity instruction set architecture (ISA). Usually these cores are provided by FPGA manufacturers and semiconductor companies. Experimental or educational cores are available in open-source. In particular, tailor-made embedded systems are based on commodity processors. They benefit from the mostly well-supported tool-chains of these processors.

The here presented approach is based on a hardware extension of a soft-core processor for TSSs. Due to the fact that the design size of the silicon devices and the used logic cells (LCs) of an FPGA device have a direct impact on energy consumption, the extension should be moderate in its size. Table 2.1 gives an overview of some typical soft-core processors for TSSs. Thus soft-core processors utilize a small amount (less than 10 %) of LCs of typical FPGAs. In comparison to commodity processors their design size in silicon devices is very small. For example the IHP430X MCU (see Table 2.1) occupies $24.7\,\mathrm{mm}^2$ in a $0.25\,\mu\mathrm{m}$ silicon device [PSB+13]. It is five times smaller than a 15 years old *Intel Katmai Pentium® III* processor, which has a size of $128\,\mathrm{mm}^2$ in the same technology.

Table 2.1: Design size of soft-core processors synthesized for an Altera FPGA.

Processor	ISA	Address space	LCs	FPGA
TinyVLIW8	proprietary	11-bit	1,342	Cyclone II
openMSP [Gir10]	MSP430	16-bit	2,841	Cyclone II
IHP430X	MSP430X	20-bit	4,107	Cyclone II
ARM Cortex-M1 [plc14]	ARMv6	32-bit	2,600	Cyclone II
LEON2 [For04]	SPARCv8	32-bit	9,299	Cyclone

2.5 Examples of tiny scale applications

As introduced at the beginning of the thesis, an application of a TSS executes small-scale software without multi-user support and very often without any user interface. Most of the TSS applications can be partitioned in a sense-and-control component and a communication stack. Both parts are usually controlled by a small glue code, which can be seen as an application. Figure 2.3 illustrates such a TSS application.

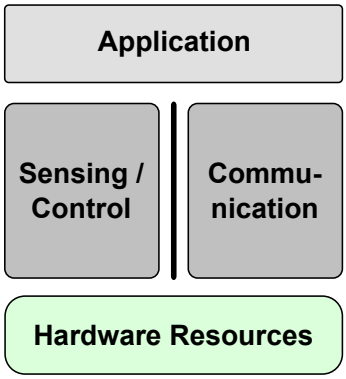

Fig. 2.3: Block diagram of a TSS application. A TSS application can be partitioned in a sense-and-control component and a communication stack. Both parts are functional separated and controlled by the application's glue code.

Even though such a simple application can be implemented on a platform without multi-user support, at least three individual software activities[2] can be identified. Hence, in a well-defined implementation the sense-and-control module, the communication stack, and the application must be separated. It can be assumed that the sense-and-control module and the communication stack do not share any data. Instead, the application's glue code acts as a bridging component that processes the incoming or sensed data and performs a selective data forwarding. A secure isolation of these SAs can be guaranteed on a system with a proper memory protection only. Such a protection has to ensure that an activity has access only to those data that it owns as well as to public interfaces of other software activities. This cannot be guaranteed on TSSs with insufficient memory protection capabilities.

Although the isolation of the communication stack increases the system's security level in a significant manner, TSSs may ask for a more fine-grained separation as well. Therefore, in the following subsections two tiny scale applications are introduced. These applications are used during this thesis as practical examples to illustrate the proposed concepts more detailed.

2.5.1 Meetering app

The *Meetering* application was implemented within the *Diamant* project at the IHP [IHP]. The major goal of the application is to predict the point of failure of greenhouse lamps. As illustrated in Figure 2.4, the lamps are organized in clusters where each cluster is controlled by a sensor node. A tiny scale application on the sensor nodes is used to log the active time of each lamp of the cluster. Furthermore, the sensor node can switch the cluster's lamps

[2] As mentioned in a previous section on TSSs a user can be substituted by a software activity.

on and off. The sensor nodes are controlled by a PC application. The PC is connected by a serial interface to a sink node. The sink node uses a wireless interface to communicate with the sensor nodes. The communication is organized by a proprietary protocol that follows the request/response method where the sink node sends a request to the sensor nodes to fetch the logged data or to transmit a command. The sensor nodes and the sink node are IHPnodes with an MSP430F5438A without any memory protection capability [PSL10].

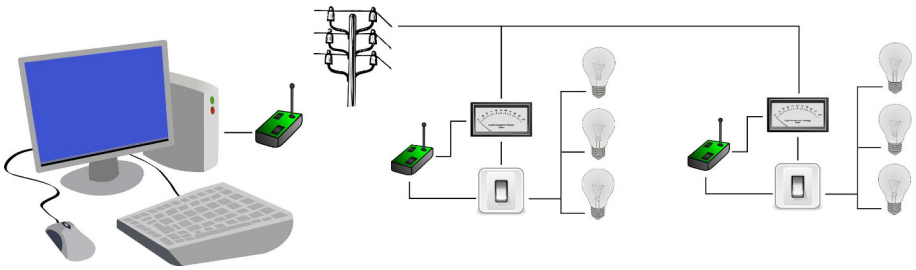

Fig. 2.4: Meetering application to control the service time of greenhouse lamps. Lamps are organized in clusters, which are monitored and controlled by a wireless sensor node.

2.5.1.1 Components, software activities and resources

The *Meetering app* is implemented in langOS and TinyOS. Both are operating systems designed for TSSs. The following description is based on the langOS implementation. Although the application may look similar in TinyOS, the software activities and the data handling details can differ. Figure 2.5 shows the modules of the Meetering App implemented in langOS. The application includes three major components: a protocol stack, a storage component, and a capture/control component.

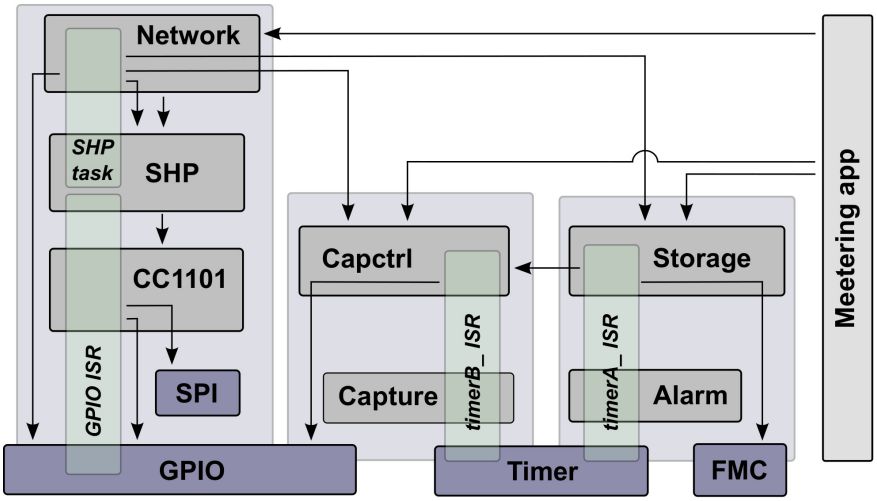

Fig. 2.5: Modules and software activities of the Meetering application implemented in langOS.

The protocol stack consists of the single hop protocol (SHP) and a CC1101 radio driver. The radio driver uses the general-purpose input/output (GPIO) module and the serial peripheral interface (SPI) module to communicate with the CC1101 IC. The SHP implements a task that takes the network packets from the CC1101 interrupt service

routine (ISR). The packet response is implemented in the network module, but will be executed within the context of the SHP task. The task sends response packets containing captured data or controls the lamp switches via the capctrl module.

The *storage* component includes a flash memory controller (FMC), which uses the MSP430's *infomem* to store the current system state. The store operation is executed within the context of the timer interrupt. Read operations are executed by the SHP task to fill the response packet.

The *capctrl* component includes a capture service that uses the timer driver to capture external events sent by the power meter. The external events are handled by the timer ISR, which stores the values in a temporary buffer. The data is written to the infomem by the storage component.

Beside the three main components the bootstrap loader is responsible for initializing the modules. The bootstrap loader is executed before interrupts and tasks are activated. Therefore, we can identify four different software activities. Since the application is mainly an event responder it does not handle a lot of data. The capctrl and storage components process only integer values and have access to dedicated peripheral registers. The protocol stack is the sole component that processes a complex data structure. langOS provides a `netpkt`-object that includes packet data as well as meta information. The object is passed by reference between the processing functions within the network stack.

2.5.1.2 Interfaces and security threats

The *Meetering* app uses the wireless interface to communicate with the sink node. As result of the shared character of the wireless medium the node can receive packets from any node in its transmission range. Furthermore, packets can be eavesdropped and manipulated. Therefore, the sensor node needs special care while processing network packets. Unfortunately, the protocol stack is the most complex software component of the sensor node. Software weaknesses in the C implementation of it cannot be completely excluded.

Even though the greenhouse may not be a CI, applications of CIs are similar and are exposed to similar security threads. In the greenhouse scenario an attacker could try to do the following:

- fake the captured service time of lamps,
- report a wrong lamp status,
- switch lamps off or on, or
- sent response packets without a request.

With a look on the ”famous” *Stuxnet* attack, we can recognize that very similar attack goals had been achieved. To destroy or to disturb the uranium enrichment the centrifuge controllers were attacked and manipulated, so that they drove the centrifuges out of their specification and returned faked status messages to the central controlling unit.

Traditional approaches introduce cryptographic mechanisms to protect the wireless communication. Furthermore, filter technologies and anomaly detection systems can be integrated to analyze the network packets. But all these technologies do not really provide security for the open wireless interface. Instead more complex software modules with possible software weaknesses become part of the system, so that the probability of local attacks may be increased. A secure isolation of the four software activities can enforce that the vulnerable components, mainly the modules of the network protocol stack, are separated reliably from the valuable components.

2.5.2 A secure wake-up receiver (SWUR)

In battery driven wireless applications a radio transceiver is one of the most significant power sinks. Especially since the power consumption is similar when sending and receiving, power-centric applications reduce the active time of the transceiver by using low duty cycle protocols, which switch the transceiver off. To overcome the drawbacks of these protocols in recent years significant research was done in the area of power efficient wake-up receivers [SBS02, DEO09]. These ICs consume only few micro watts when awaiting a wake-up signal. But wireless wake-up receivers are vulnerable against depletion attacks in which a wake-up signal is sent repeatedly to deplete the mote's power supply. For the prevention of a simple replay of a wake-up sequence we implemented a SWUR that extends a common wake-up receiver with a security module that features a modified TOTP algorithm [SKM14, MMPR11]. The system is illustrated in Figure 2.6.

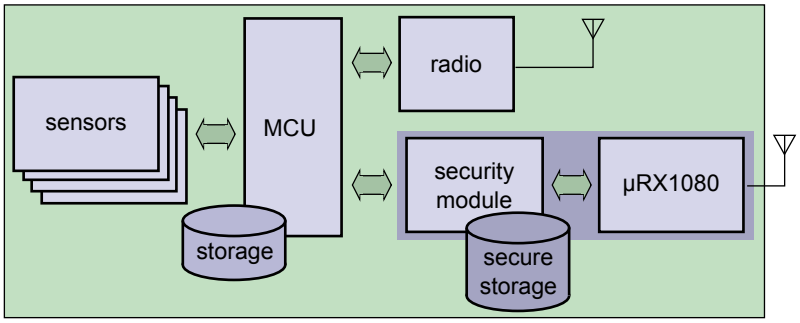

Fig. 2.6: Block diagram of a wireless sensor node with a SWUR.

The μRX1080 wake-up receiver was developed by Fraunhofer IIS and is able to detect two 31-bit codes by less than 3 μW power consumption [Fra10]. The reception of a code is signaled by a physical line, which can be used to trigger an interrupt on the MCU. We integrated a security module between the MCU and the μRX1080, which expects a sequence of codes that forms a one-time password. The one-time password is generated by the TOTP algorithm, which is based on the hash-based one-time password (HOTP) algorithm [MBH$^+$05], and employs a time-synchronized SHA-1 keyed-hash message authentication code (HMAC). The algorithm is completely implemented in the security module. The module can be configured via an SPI. Via the interface, the MCU has limited access to the TOTP counter value for resynchronization purpose and to the TOTP values to send a wake-up signal by its main radio in a multi-hop scenario.

The secure wake-up scheme ensures that an attacker cannot wake-up a sensor node without the knowledge of a valid authentication pattern. Therefore, the protection of the TOTP algorithm's data are crucial. The security module is similar to a security co-processor. It is

currently tailor-made to implement the TOTP algorithm. Nevertheless, reasoned by its soft-core processor it can be used for further applications, where a hardened soft-core processor is necessary to provide a high level of security.

2.5.2.1 Security module of a secure wake-up receiver (SWUR)

The security module implements basically the TOTP algorithm and is equipped with a soft-core processor supported by a hardware-based timer and a SHA-1 function block. The IC provides a secured interface to communicate with an external MCU. The interface follows the request-response method. The MCU must send a request command to the SWUR to initiate an action or to get any information. The SWUR has an interrupt line to signal an internal event. The event may trigger a request operation on the MCU. The functional scheme of the SWUR is shown in Figure 2.7.

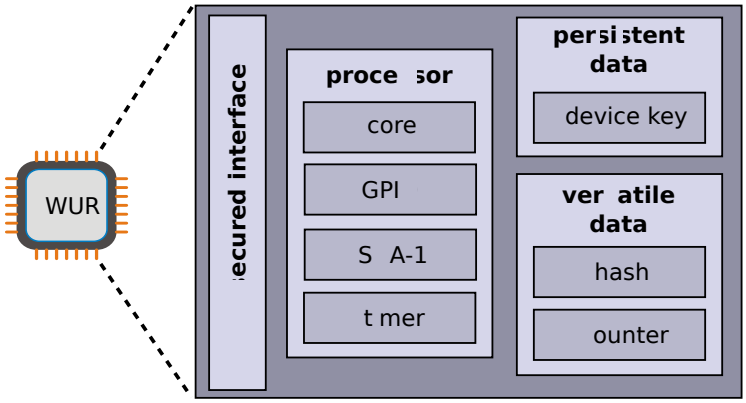

Fig. 2.7: Functional schema of the security module of the SWUR IC.

The basic structure of the SWUR scheme is similar to a trusted platform module (TPM) proposed by the trusted computing group (TCG) [ZDB09]. Beside the processor with its peripherals the IC is equipped with a persistent and a versatile memory. The persistent memory stores a device key that cannot be modified or read via any external interface. The key is used to generate the TOTP hash. The hash is based on a counter value that is periodically incremented by the processor. The counter value can be updated via the external interface for re-synchronization with other sensor nodes. Furthermore, the SWUR provides read access to the counter and the hash value. Both read operations return only a subset of the values.

The soft-core processor implements three software activities:

The *key management* implements the security critical core of the module. It is periodically invoked to perform a HMAC operation to update the TOTP hash values. Therefore, the SA needs read access to the private key and full write access to the counter value and the hash value.

The *host interface* is the public interface of the security module. It processes requests received via the public interface and generates the response information. The

SA needs read access to TOTP hash values and read and write access to the counter value. Because of its exposed position it is the most vulnerable SA.

The *symbol decoder* decodes the signals provided by the wake-up receiver. The SA is responsible for configuring the wake-up receiver during a symbol reception. The SA has no external interface and is controlled by the hardware-based symbol decoder. It needs read access to the TOTP hash values.

The application has three event sources: timer, GPIO, and SPI, which trigger the three SAs. The events are mostly handled fully within the corresponding SA. Hence, a control flow switch between SAs is not essentially necessary. But by reason of the limited resources of the soft-core processor the SAs share large objects.

2.5.2.2 Security threats

The SWUR IC may be used as a sealed storage on a wireless sensor node and provides a trustworthy wake-up signal to the MCU. On commodity wireless sensor nodes we have to assume that the MCU does not provide any security capabilities. Therefore, the MCU's interface must be seen as a public one. An adversary may hi-jack the MCU by a remote attack. Even under these circumstances the SWUR must be able to continuously provide the following primitives:

- a secure storage of private data (private key, counter value, and TOTP hash),
- continuous operation of internal functions (periodical TOTP hash update and wake-up signal detection), and
- an enforcement of a SWUR endorsement, when generating a wake-up signal.

Based on these primitives we can form the security threats for the security module of the SWUR. The module is basically open by its command interface. Therefore, it must be prevented that commands can cause any malicious manipulation or leakage of security information or having impact in the internal operations for hash update and symbol reception.

CHAPTER 3

The art of resource isolation

Computer security is the protection of computing services and the data that they store and access. The basic motivation for isolating system resources is to keep an activity's malice or error from harming other activities. Such a harm can be inflicted by an activity in several ways:

a) by modifying or destroying data of another activity,
b) by reading data of another activity without permission, or
c) by degrading the system's service.

This chapter gives a brief overview of resource separation and access control techniques. The first major operating system to be designed as a secure system was Multics [CSC72]. It influenced future operating systems such as the Unix operating system family. We start the overview with general introduction in access control. Afterwards, we present memory separation schemes, which are fundamentals for future, higher-level memory protection schemes. We will conclude the subsections with operating systems for TSSs and their security features and extensions.

Traditional resource isolation on desktop or server computer systems focused on control based on the identity of the user running the program. But this approach has taken the view that the program itself is trustworthy. By the increasing complexity of modern computer programs this view cannot be sustained. In our definition of TSS a user context is even not given due to the absence of user in general. Hence, our focus is on software components, called a software activity, that accesses data of its and of foreign protection domains. In which we define a protection domain as follows:

> **Protection domain** defines the private data, code and stacks that an application can access, along with any data shared with other domains [KCE92]. Software activities of TSSs are executed within the context of protection domains.

3.1 Access control

In general, access control is the selective restriction of access to places or resources. In the context of computer security, access control includes the essential services of *authorization*, *identification and authentication*, *access approval*, and *accountability*. Authorization is the service that determines which rights are assigned to a subject. Identification and authentication include the service of finding out who someone is or what something is and the service to verify the truth of this identification. Access approval is the service that grants or denies access operations. It includes a function that compares the authorization with the access request. Accountability is the service that identifies what a subject did.

3.1.1 Access matrix

In order to provide a facility to control activities in computer systems, it is necessary to have a systematic way to control access of one activity to another one. Such a process access control can be simply handled by tree structures [Han70]. In addition, a useful convention for sharing among activities must be provided to have a systematic way of describing what is to be shared and of controlling access to shared things from various activities. Both can be handled by a more general machinery, which is called *object system* [Lam71]. The object system has three major components: a set of *objects*, a set of *domains*, and an *access matrix*.

An access matrix or access function A, as illustrated in Table 3.1, determines the access of domains to objects. In the matrix a row is labeled by a domain name and a column is labeled by an object name. The choice of objects is a matter of convention, which is determined by the requirements of the system. Generally, it must be guaranteed that in each system the object names are globally valid.

*Table 3.1: An example of an access matrix (*copy flag set).*

	Domain 1	Domain 2	Domain 3	File 1	File 2	Process 1
Domain 1	*owner control	*call	*call	*owner *read *write		
Domain 2		*owner control	call	*read	write	wakeup
Domain 3			*owner control	read	*owner	

An element A_{ij} of the access matrix determines the access rights of domain i to object j and consists of a set of access attributes, which are typically strings, as 'read', 'write', or 'owner'. In addition, at each attribute a *copy flag* can be attached. The copy flag controls the transfer of access rights. According to the example of Table 3.1 a domain d_n can modify the list of access attributes for domain d_m and object x as follows:

a) d_n can remove access attributes from $A_{d_m x}$ if it has 'control' access to d_m.
b) d_n can copy to $A_{d_m x}$ any access attributes it has for x, which have the copy flag set, and can say whether the copied attribute shall have the copy flag set or not.
c) d_n can add any access attributes to $A_{d_m x}$, with or without the copy flag, if it has 'owner' access to x.

The copy flag is required to control that a subordinate domain does not wantonly give away access to objects. The rules above do not permit the 'owner' of an object remove access attributes. If it is permitted an additional rule is appropriate

d) d_n can remove access attributes from $A_{d_m x}$, if d_n has 'owner' access to x, provided d_m does not have 'protected' access to x.

The major concern when implementing an access matrix is their sparse utilization. Most elements will be left empty in real systems. A more efficient alternative is the array of triples $< d, x, A_{dx} >$. A triple look-up is started whenever the value of A_{dx} is required. Due to the implementation of this alternative is also impractical for a number of reasons, a more efficient implementation can be obtained by using a list of objects, which can be accessed by a domain, or list of domains, which can have attributes for an given objects.

Although a usage of an access control matrix is not common in modern computer systems it is used in systems with high security demands. For example the smart card MCU ST16SF48A implements a memory access control matrix (MACM) to set-up user-defined access rules from any memory sector to another one [STM00].

3.1.1.1 Access control lists (ACLs)

The list of domains, which have attributes for given objects, is usually stored in an ACL. It is similar to a column of an access matrix. An ACL is a set of pairs $< d, A_{dx} >$ for a given object x. The usage of an ACL is often more efficient due to the fact that a pair is needed only if element A_{dx} is not empty.

An ACL_x is usually tightly coupled with the object x. Modern operating systems use ACLs to control access to file system objects. Each object x of the file system has an ACL that contains a subject, a user identified by a user id, and file operations, as 'read', 'write', or 'execute'. In case of a user access on a file system object the attached ACL is searched for the object with the given user id. If it exists, the access rights are compared with the demanded operation. If it is not matching or an object is missing the access will be denied.

The application of ACLs is useful, if it can be easily coupled with the object and the number of subjects/domains with different attributes for the object is quite small. In case of many different domains for a single object the implementation becomes inefficient. Therefore, ACL implementations use groups or wildcards to reduce the number of objects. In the context of memory protection the usage of ACLs is quite uncommon because the management becomes very intensive in dynamic systems. Static systems can benefit from ACLs because the number of domains that have access attributes for an object is usually quite small.

3.1.1.2 Capabilities

In contrast to an ACL, which is object related, capabilities specify the access attributes of a given domain. The capabilities C_d are a row of an access matrix and contain a set of pairs $< x, A_{dx} >$ for the domain d. Similar to an entry of an ACL, a capability is only required if $A_{dx} \neq \{\}$.

Capabilities are coupled with a domain. In case of an access to an object the capabilities of the active domain must be determined and searched for a corresponding entry. The application of capabilities is useful if the number of objects for each domain is small or the lifetime of the objects is short. This is usually given for memory protection in dynamic systems. Processes (domains) will allocate and free memory resources (objects) very frequently. Therefore, each process holds a list of memory resources, which it owns or is readable, writable, or executable for it, see Figure 3.1.

Even more common computer systems do not provide fine-grained capabilities for memory protection. On these systems very simple protection schemes are implemented in hardware. A brief introduction will be given in Section 3.2.3. Saltzer et al. describe a concept of hardware capabilities and ACLs. He has observed that systems combine both in order to offer a blend of protection and performance [SS75]. More fine-grained protection mechanisms based on capabilities are implemented by the Mondrian Memory Protection or CHERI [WCA02, WWC+14].

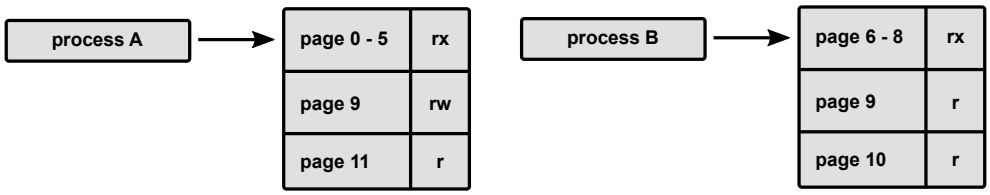

Fig. 3.1: Process capabilities for memory protection. Each process holds its own capabilities with memory resources and access attributes.

3.1.2 Basic models of access control

The standard *"Trusted Computing System Evaluation Criteria (TCSEC)"* of the US department of defense, also known as the *Orange Book*, describes two fundamental protection schemes: discretionary protection and mandatory protection [Uni85].

3.1.2.1 Discretionary access control (DAC)

The discretionary *"need-to-know"* protection provides a separation of subjects and objects. It enables subjects to be able to protect private information and to keep other subjects from accidentally reading or destroying their objects. Furthermore, it is allowed to authorized subjects to change the access control attributes of their objects. Therefore, a subject is able to specify whether other subjects have access to objects. The access control management is based on subjects that are object owners. A central controlling instance is not involved.

The relation between subjects and objects is represented by an access matrix and access is restricted to objects based on the identity of subjects. In real systems, ACLs or capability-lists are used to overcome the concern of the usually sparse matrix utilization.

A simple form of discretionary access control (DAC) can be seen in the basic file permissions (read, write, and execute) of a Unix system or file passwords, where access to a file requires the knowledge of a password created by the file owner. But the DAC has the basic weakness to fail to recognize a fundamental difference between human users and computer programs, which makes it uncommon for systems with fine-grained security demands.

3.1.2.2 Mandatory access control (MAC)

MAC permits the specification of policies limiting the interaction between subjects and objects. It is more difficult to handle than DAC but overcomes its limitation in a fine-grained access control. It is usually used to protect high-sensitive information. The mandatory access control (MAC) mechanism requires that subjects and objects are labeled with a unique identifier to allow policies to be written. Furthermore, it also requires a broad set of enforcement points across the majority of operating system operations.

MAC mechanism can be applied by two methods:

Multi-level security (MLS) systems are based on security levels. Each object is assigned to a security level, which separates the objects in ”horizontal” security layers. Within a security layer information flows are not restricted. But it is denied to exchange

information between different security layers. Especially a layer with a higher security level may not lack any information to a layer with a lower security level. Subjects are also assigned to security levels. A subject has access, so called *clearance*, to an object if its security level is equal or higher than the object's level [BL73].

Lattice-based access control guarantees secure information flow in a computer system between any combination of objects and subjects. The central component is a lattice structure derived from the subjects and objects and justified by the semantics of information flow. The model provides a unified view on all systems that restrict information flow and enables a classification of them according to security objectives. It defines subjects S, objects O, security classes SC, a class-combined operator $\oplus$, and a flow relation $\rightarrow$. The security classes are closely related with the concepts of "security classifications", "security categories", and "need to know". Each object and each subject is assigned to a security class. The $\oplus$ operator defines how to label information obtained by combining information from two security classes. The flow relation $\rightarrow$ is defined on pairs of security classes. For classes A and B, we write $A \rightarrow B$ if and only if information in class A is permitted to flow into class B [Den76, San93].

Only few computer systems implement MAC, examples are Trusted Solaris, TrustedBSD [Wat01], and SElinux implementation [LS01]. A brief introduction of SElinux is given in Section 3.4.2.

3.1.2.3 Role-based access control (RBAC)

The RBAC concept began with the multi-user and multi-application on-line systems pioneered in 1970's [FD92]. The RBAC model was introduced by Sandhu and has been emerged as the leading standard for defining access control constrains [SCFY96]. The National Institute of Standards and Technology (NIST) RBAC model is defined in terms of the four model components: core RBAC, hierarchical RBAC, static separation of duty relations, and dynamic separation of duty relations [SFK00].

The core RBAC illustrated in Figure 3.2 embodies the essential aspects of RBAC. The basic concept is that users are assigned to roles, and users acquire permissions by being members of roles instead of getting access to an object in a traditional access control system. Furthermore, RBAC includes a requirement that user-role and permission-role assignments can be many-to-many. In addition RBAC supports the concept of sessions. A session is a mapping of a user and an activated subset of assigned roles.

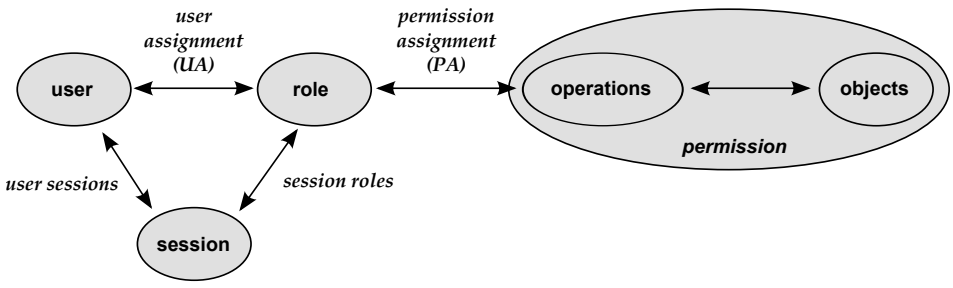

Fig. 3.2: The core RBAC model.

A permission to an object is defined as an operation applied to an object. These operations and the objects are dependent on the type of the system in which they are implemented. An operation is an execution of a specific function. This can be a data manipulation function as well as a simple view operation. An object is an entity that contains or receives information or a system resource.

An additional concept is the hierarchical RBAC, where the two exclusive types *limited* and *general* exist. A further concept is the constrained RBAC that defines the static separation of duty relations and the dynamic separation of duty relations.

3.1.2.4 RBAC in wireless sensor networks

Wireless sensor and actor networks can be seen as distributed systems that perform a specific sensing, monitoring, or acting task. The network consists of sensor and acting nodes, gateway nodes, and a network sink. In some networks multiple sinks are available. Due to the limited resources of the individual node, usually a specific task was assigned to each node. We have already mentioned security problems in this type of networks in Section 2.1.2.1. A secure transmission of membership lists and key information is proposed by Perrig et al. [PST+02]. Integrity and confidentiality in WSNs are provided by TinySec, a link layer security protocol for TinyOS [KSW04].

Moon et al. propose that a more flexible authentication and authorization framework can be achieved by using the alternative access control methodology task-role based access control (T-RBAC) [MKP07]. The T-RBAC model is based on the concept of the classification of tasks. It deals with each task differently according to its class and supports task level access control and supervision role hierarchy [OP03]. Moon et al. sketch that T-RBAC modules can be integrated in sensor nodes and have user role assignments. In an adaptation of the T-RBAC model on WSNs a *user* means a sensor node and a *role* means an assigned role of each sensor node. Furthermore, T-RBAC defines *tasks*, which were adapted to use resources of a sensor node.

3.2 Memory separation schemes

Memory separation in the context of operating systems denotes the property that a certain software entity can access only those resources, which are assigned to itself. To guarantee security, separation has to ensure that it is still effective in case that a software entity is running malicious code. Multiple independent levels of security are based on the concepts of separation [Rus81] and controlled information flow.

In this section we will introduce classic hardware-based separation concepts. They are mainly realized and used on powerful systems, but known to be very effective. The last section gives a brief overview regarding MPUs tailor-made for resource restricted devices.

3.2.1 Message systems

A message system is an idealized system to illustrate the meaning of the term ”protection domain”. In such a system activities share nothing and communicate with each other by means of messages only.

3.2.1.1 Messages

A message of a message system consists of any kind of a sender’s identification and a chunk of data. The identification can be an integer number as part of the message or the number of an incoming channel. It is important that this number is assigned by a trusted instance, so that it cannot be forged [Lam71].

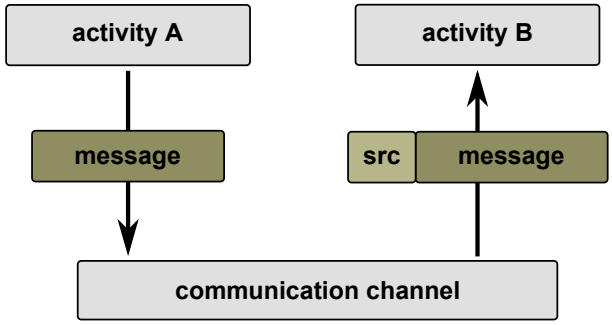

Fig. 3.3: Communication in a message system requires that a trusted instance inserts the source into the message.

In a message system any activity can send a message to any other activity. The messages are received in the same order they were sent. In such a message system, everything belongs to an activity and cannot be accessed by an activity other than its owner. Hence, each activity is a single domain, whose resources are protected by isolation. It is similar to a separate machine with its resources, e.g. memory, which are isolated by hardware except for the message transmission subsystem. The message system provides a locally complete protection system. However the system can provide a ordinary subroutine call in which a process (A) is calling a function of process (B), in a way that A sends a message to B. This system works if even B must be protected from A, for example, if B is a supervisor of A, A can ”enter” B, namely at the point where it waits for A’s messages. A random transfer (enter of B) to an arbitrary point in B is not possible. Furthermore, the ”return” is protected as well. Thus, if A mistrust B, B will not be able to return to A except in the manner intended by A [Lam71].

3.2.1.2 Marshalling

Based on communication channels between activities, the mechanism of a subroutine call can be emulated in a way that an activity *A* sends *B* a message specifying the parameters: in, out, and inout. After sending a message to invoke a subroutine of B the sender waits for a reply message from B, if required. The reply will be sent by B with another message. It is important that the message structure is globally defined, so that each peer can identify the types of the different parameters. In detail, the caller must execute the following steps:

- A request message is constructed, which contains all input and inout parameters and a key that identifies the service on receiver's side (*marshalling*).
- The message is send to the server.
- The inout and out parameters are read from the reply message (*unmarshalling*).
- The result is returned to the caller.

The callee usually implements a loop to wait for incoming requests. In detail a callee will execute the following:

- It receives the request message and uses an included key to determine the addressed service.
- The service extracts the in and the inout parameters form the received message (*unmarshalling*).
- It executes the service function/method with the extracted parameters.
- Afterwards, it constructs the reply message and stores result values in the `inout` and `out` parameter (*marshalling*).
- send the reply message back to the caller.

The marshalling performance is a crucial issue of message systems. Especially logically completely protected systems with small software activities, e.g μ-kernel systems, benefit from fast marshalling techniques [Lie95b]. In these systems, interface definition language (IDL) compilers are used to generate the caller and the callee stub automatically. Afterwards, a user can modify the generated code to implement additional features [HLP+00].

3.2.2 General-purpose memory management

In a commodity system memory protection is implemented by privilege levels and hardware-based resource isolation. "Protection" is a general term for all mechanisms that control the access of a program to other things in the system [Lam71]. It enforces that a software entity is capable to operate unrestricted on its resources and in addition that foreign resources are invisible to it or protected from it.

Early memory protection starts with the protection key scheme introduced with the IBM360 system [IBM64]. It partitions the system memory in regions with a fixed size and assigns a single key to every region. If a process generates a memory access the key currently stored in the CPU status register is compared with the key assigned to the memory region. If the two keys are equal, the access is granted. Otherwise a protection exception is triggered. It is a simple and effective scheme, but it does not support shared regions and does not check memory accesses generated by an instruction fetch.

3.2.2.1 Protection rings

Hierarchical protection domains, often called protection rings, are a mechanism to protect high sensitive information or privileged functionality from lower privileged functions. General-purpose computer systems provide different privilege levels to control access to security and safety critical resources. These r rings are named by integer numbers from 0 through $r-1$. In such a system, the access capabilities of ring m are a subset of those in ring n whenever $m > n$.

The x86 microprocessor architecture provides four different privilege levels, when used in the protected mode. As illustrated in Figure 3.4, the x86 levels are called rings and start with ring 0, the highest privileged, and go up to ring 3, the least privileged. The rings are used in general purpose operating systems to separate the system kernel from user applications. Kernel information can be accessed only if the accessor function is executed in ring 0. Applications executed in ring 3 cannot access directly any kernel information. In diametrical opposition kernel functions can access any application information. Although the x86 microprocessor supports four different levels, most operating systems use only two of them. The rings one and two are usually unused, except OS/2, which uses ring two for privileged applications with I/O access permissions.

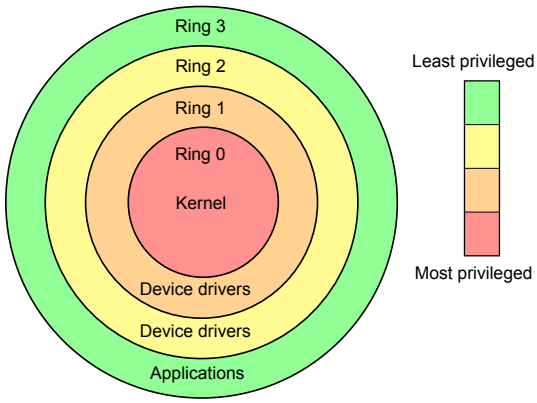

Fig. 3.4: Privilege rings for the x86 available in protected mode [Com07].

Beside the x86 architecture, protection rings were implemented in the MULTICS system, a highly secure predecessor of the today's UNIX operating system family. This OS, implemented on a Honeywell 645 computer system, supports a segmented virtual memory with a limited set of access control mechanisms. Therefore, a software approach to enforce protection rings was necessary. Beginning with the Honeywell 6000 series, a new processor with an improved set of access control mechanisms, which implements rings almost completely in hardware, was introduced [SS72].

Recent microprocessors provide an additional protection ring beyond ring 0. It was introduced to improve virtualization of x86 operating systems without static or dynamic modifications. The new ring, called *ring -1*, is a ring higher privileged than ring zero and makes a separation of operating systems possible. More information regarding virtualization will be given in Section 3.3.5.

3.2.2.2 Segmentation

When using an MMU the memory can be divided in memory segments. Segmentation is basically used for implementing virtual memory and memory protection. Therefore, to each segment an individual base address and access rights can be assigned. Applications use virtual addresses that include a segment number and an offset instead of a physical address. During address translation, necessary on each memory access, the real memory address can be calculated and the demanded access type can be checked against the stored access permissions. The translation operation itself is performed by a hardware MMU.

Since the segment size can be set individually, it corresponds usually to the memory allocation of programs or data tables. But the individual size of segments causes a memory fragmentation on dynamic systems, where segments are frequently allocated and freed. Therefore, the more flexible paging concept is mostly used on commodity systems. Some architectures, as x86, support a combination of both.

3.2.2.3 Paging

The most common memory organization in general-purpose computer systems is the paging concept. The key feature of the paging concept is the mapping of virtual address spaces to physical memory resources. A virtual address space is the program's view on memory resources. In commodity operation systems each process gets its own virtual address space. It may include shared areas with individual access rights. By using paging the contiguous virtual memory can be mapped on fragmented physical memory as well as partial inactive sections. Hence, the physical resources can be managed more efficient than directly accessed memory. The mapping is usually done by a hardware MMU. Beside the memory mapping, the paging manages memory access rights. Access rights can be set individually for each page. On x86 systems, they include read/write access, user/supervisor mode, and non-execution bits. The access rights are checked by the MMU as well. In case of an access violation an interrupt is raised and the mapping is not resolved.

Multi-level page table

The page size is typically in the range from 1 KiB to 2 MiB. Depending on the size of the address space the number of pages to manage differs from few entries, 16-bit address space, to a multiple of million of entries, 64-bit address space. Hence, the pages are organized in multi-level page tables. A schematic illustration of a single level page table is shown in Figure 3.5. The virtual address is divided in a page table offset and page offset. The base address of the page table is individual for each process. A page table entry addressed by the page table base address and the page table offset includes the physical base address and status bits, which includes the page access rights [Tan09].

For an efficient storage of page table entries the address space is usually managed by multi-level page tables. In a multi-level page table the virtual address is divided in $k \times n$ bits for the k page tables and m offset bits. The page table entry of the k-th level contains the base address of the page table $k + 1$. The last level contains the physical base address. As an advantage of multi-level page tables currently unused tables can be swap out [Tan09].

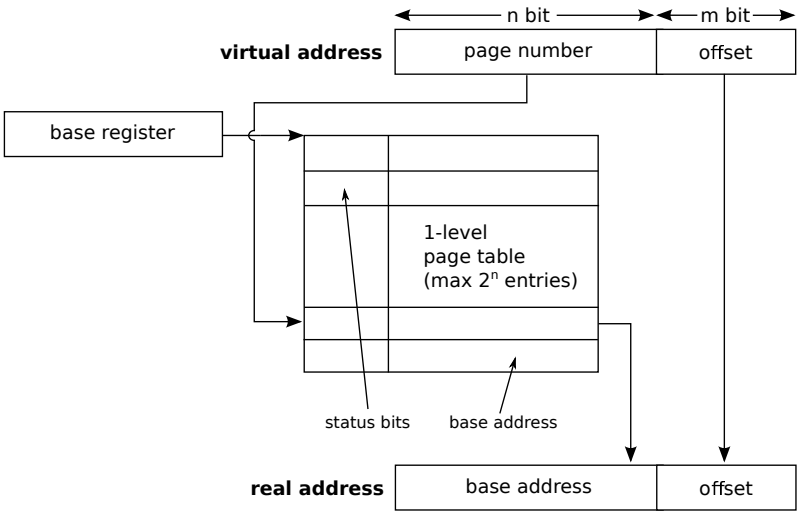

Fig. 3.5: Diagram of a single level page table (Wikipedia).

Inverted page table

In large address spaces multi-level page tables need a huge amount of memory and will be used sparsely only. The problem can be avoided by using an inverted page table. An inverted page table stores the virtual address instead of physical address. On a page look-up the table must be traversed to find the entry containing the virtual address. In case of a match the entry index is combined with the base address to build the physical address. Especially in large tables a complex search algorithm is required. Usually a hash table is used to speed up a page table look-up.

Guarded page table

The key idea of guarded page tables is to augment each page table entry by a bit string g of a variable length, which is referred to as a *guard*. The translation process starts in the same way as a multi-level page table look-up. The selected entry however contains not only a pointer but also the guard g. If g is a prefix of the requested virtual address the translation process continues with the remaining postfix or terminates with the postfix as page offset. Figure 3.6 presents the look-up of a 20-bit address by three page tables [Lie95a].

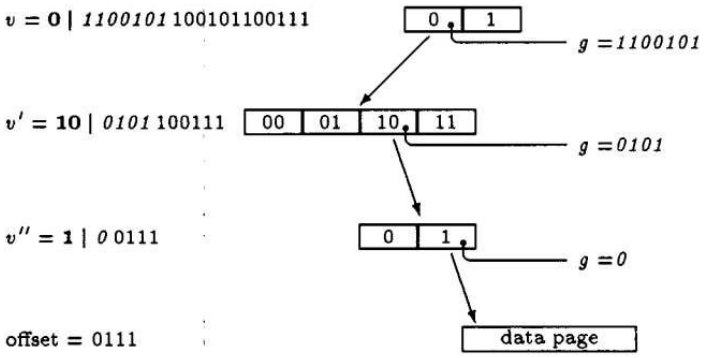

Fig. 3.6: Guarded page table tree [Lie95a].

In contrast to conventional multi-level page tables guarded page tables have a much higher density. The multi-level page tables need a huge amount of page table entries for non-mapped pages to build the tree and inverted page tables need a complex look-up scheme. Guarded page tables avoid these problems by working as a multi-level page table, but skipping empty entries of the intermediate page table levels. Furthermore, the page table size can be mixed. All powers of two are admissible. The same holds true for the size of the pages. All this makes the concept attractive for large or sparse used address spaces.

Translation look-aside buffer (TLB)

Since multi-level page tables as well as inverted page tables require a time-consuming page look-up scheme, modern processors use a cache memory for recently used entries. This cache memory, the translation lookaside buffer (TLB), is usually implemented as a content-addressable memory (CAM), also known as associative memory, with a limited number of entries. CAM compares input data against stored data and returns the matching address or the matching data, in case of an associative memory. If the requested address is present in the TLB the stored physical address can be used to access the memory. In case of a TLB miss, the requested address is not stored in the TLB, a page table look-up is required. After the page table entry is determined the entry is entered into the TLB. Due to the fact that each process has its own virtual address space, TLB entries become invalid in case of a process switch. Although software TLBs make a selective flushing of TLB entries feasible, most hardware TLBs do not support this operation and require a full TLB flush.

3.2.3 Capability-based computer systems

We have already introduced the concept of capabilities in Section 3.1. In computer systems a capability-based system differs significantly from conventional systems. We defined a capability as a token or key that gives the owner permissions to access an object. In a computer system it is implemented as a data structure that contains a *unique identifier* and *access rights*, as illustrated in Figure 3.7. The identifier addresses the object, which can be any logical or physical entity, such as a portion of memory, a file, a message port, or a register.

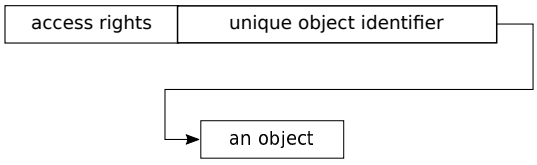

Fig. 3.7: Data structure of a capability [Lev84].

In a capability-based system each activity, e.g. a user, a program, or a procedure, has access to a list of capabilities. To perform an operation the activity might call the following:

```
ENABLE(led_capability, 1);
```

The call serves two purposes: first, it identifies the resource to be used and second, it specifies the operation to be performed. Capabilities are the basis for object protection. An activity cannot perform an operation unless it has a capability in its capability list. It is important that it must be prohibited for the activity to modify its capability list, otherwise it can access any

object. Therefore, only a trustworthy component, usually the operating system or a hardware unit, can modify the capability list of an activity. However, an activity can invoke the trustworthy component to obtain a new capability.

Although capabilities can control access to many different types of objects, early capability-based systems have used capabilities for memory addressing only. In such a system each process has a capability list that defines the memory segments it can access. Instead of using a segment table, see Section 3.2.2.2, a capability addressing system makes a direct use of capabilities as an address [Fab74]. A hardware-based address translation is based on capability registers, whose content is loaded from the capability list and stored in the main memory by using special instructions. A simplified hardware model of a capability-based memory system is shown in Figure 3.8.

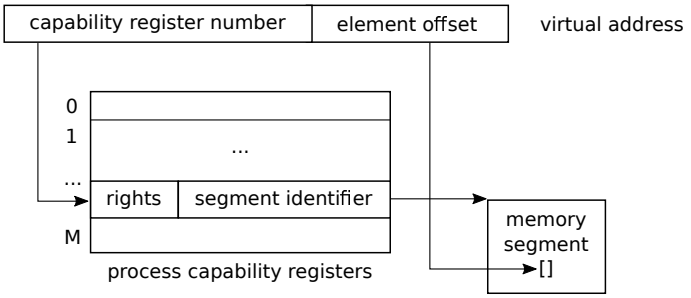

Fig. 3.8: Simplified hardware model of a capability-based memory protection system [Lev84].

The model has the following properties: A memory segment can be accessed if a capability for that segment has been loaded into a capability register. Loading a capability register is an unprivileged operation. It must be guaranteed that only a valid capability can be loaded into a capability register. The address space of a process changes whenever the program changes the capability registers. A process can share a segment by copying or sending a capability to the capability list of another process [Lev84].

A very efficient implementation of a capability system is presented by Carter at al. [CKD94]. It uses guarded pointers that identify a byte in the virtual address space, the segment containing that byte, and the set of operations permitted on the segment. The approach eliminates the indirection and related performance penalties associated with capability based systems.

3.2.4 Memory protection units

A comprehensive overview about MMUs is given by Jacob et al. [JM98]. He identifies a run-time overhead of ten to thirty percent caused by MMUs in comparison to unprotected systems. Therefore, TSSs with their limited resources and squeezed system performance include a much simpler MPU instead of an MMU. An MPU provides basically memory separation without implementing a virtual memory layer. Applications are linked and mapped to direct addresses, which simplifies the memory scheme in a significant manner. The implementation of memory protection without address translation simplifies the hardware unit in a significant manner and makes the design footprint much smaller.

A survey of hardware-based MPUs in deeply embedded systems is given by Lopriore [Lop14] and in the Ph.D. theses of Stilkerich [Sti12] and Rengaswamy [Ren07]. In the following we will give a short essay about MPUs in TSSs. The essay is not all-embracing, this would exceed

the scope of this thesis. We introduce only some examples and skip the more powerful processor core as the ARM11 or commercially available soft-core IP, e.g. the ARC core, the LEON-2, or the Nios II processor. Most of these soft-cores provide a configurable memory protection scheme, which ranges from a simple MPU to a fully featured MMU, with a more or less significant design footprint overhead.

3.2.4.1 Infineon embedded processors

A region-based memory protection is provided by the Infineon embedded processors. The Infineon Tricore TC1796 features a heterogeneous MPU, which provides two sets of region registers. One set for two code regions and one set for four data regions. Execute permissions are available for code regions only. Whereas read and write permissions can be granted on data regions.

A similar memory protection is provided by the Infineon XC2000 family of 16/32-bit micro controllers. The MCU is equipped with a simple MPU optimized for embedded control applications. It provides four different protection levels. The current active protection level is stored on the processor status register. A protection register set is associated to every protection level. Each set contains the upper and lower region addresses and the access permissions. On each memory access it is checked that the access is within the memory regions associated with the protection level and does not violate access permissions. In case of an invalid access the operation is blocked and a protection trap routine is executed [Inf11].

3.2.4.2 Texas Instruments MSP430

The MSP40 is an ultra-low power MCUs from Texas Instruments with a reduced instruction set computer (RISC) architecture. The MCU is widely used in WSNs and embedded controllers. The first devices had a single 16-bit address space without any memory protection. Later on the MSP430X architecture followed, which is binary compatible to the first device, but features a 20-bit address space. The MCU is available as soft-core processor [Gir10] as well as binary compatible silicon device designed by Fraunhofer IPMS [Grä10] or IHP [PBS+11]. A more detailed description of the MCU will be given in Section 5.1.1.1.

The MSP430 FR57xx family is a MSP430X MCU with a hardware MPU. The MPU protects the interface FRAM against accidental writes or execution of code from constant memory segments. In detail the MPU features include:

- three segments of variable size,
- individual access rights for each segment,
- independent access rights for the information memory, and
- password protected MPU registers.

Since each segment consists of pages the smallest size of a segment is a page. The page size is restricted to $1/32$ of the implemented memory size. An overlapping of segments is not possible. The upper border of segment n is the lower border of segment $n + 1$. Furthermore, the segment 1 starts with the main memory start address and the end of segment 3 is defined by the highest main memory address [Ins11].

3.2.4.3 Lopriore MPU

Lopriore presents a hardware/compiler memory protection unit in sensor nodes [Lop08]. The design takes advantage of a synergy between the hardware and the compiler. He proposes a low-cost protection circuitry inside an MCU of that effort complies with the stringent limitations existing in TSS, especially in terms of hardware complexity, available storage, and energy consumption.

A hardware level protection is provided by an MPU interposed between the processor core and the memory devices, volatile (RAM) and non-volatile memory (Flash/ROM). The memory space is logically partitioned into 2^n fixed size blocks β_i $i = 0, 1, ..., n-1$. The MPU contains for each memory block β_i a *block protection register* BPR_i. The register size d is equal to the number of supported basic domains $\delta_0, ..., \delta_{d-1}$. The MPU implements a simple access matrix. A software activity running in δ_j can access a memory location in β_i, for both read and write, if the bit BPR_{ij} is set. In case of an access violation, the active domain δ_j has no access permissions and an exception is raised to the processor.

The solution addresses mainly the reduction of the impact of programming errors on deeply embedded systems. A privileged mode and a protection scheme for the MPU management functions is not provided. The approach is focused on safety and enforces reliability. Security or protection against harmful programs is out of the scope this solution.

3.2.4.4 Mondriaan memory protection (MMP)

The Mondriaan memory protection (MMP) is a fine-grained protection scheme that allows multiple protection domains with flexible shared memory and exports protected views to other protection domains [WCA02]. In contrast to page-based approaches it allows an individual permission control at a granularity of single words. Similar to the occasionally resembling works of the Dutch painter Piet Mondriaan, the memory grid in Figure 3.9 can be painted with any pattern of access permissions. The MMP combines the flexibility and high-performance of a segmented architecture with the simplicity and the efficiency of linear addressing.

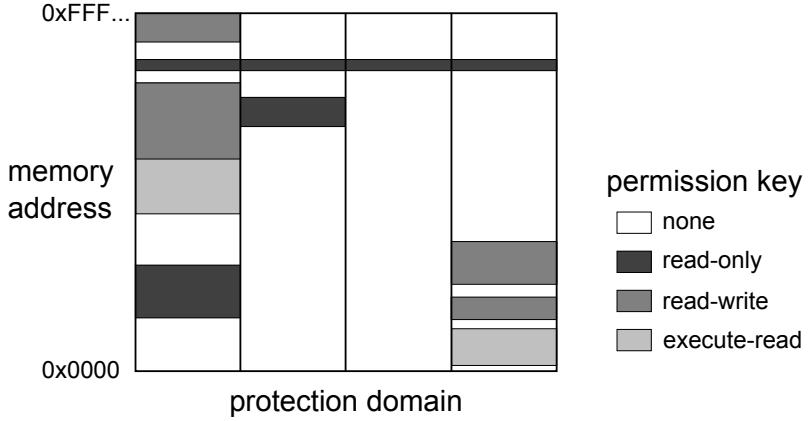

Fig. 3.9: A visual depiction of multiple memory protection domains within a single address space [WCA02].

The MMP scheme defines four different permission rights: none, read only, read write, and execute read. It can be easily modified to support more or different permission rights. Based on investigations of software examples the three basic requirements: *different*, *small*, and *revoke* are defined for a memory system.

- **Different** protection domains must be able to define different permissions to the same memory region.
- **Small** objects must be supported by the memory system. These objects can be even smaller than a single memory page.
- **Revoke** of permissions must be possible for protection domains on regions that they own.

All these requirements are supported by the MMP. A memory region is owned by a single protection domain. An owner of a region is allowed to export protected views to other protection domains. The permission rights of an exported view can differ from the original permission rights. The protection-related data of each region are stored in the multi-level permission table, where a segment is described by the triple $< base, length, permission >$. Furthermore, mini-SST entries can be defined. These entries make advantage of the sorted segment table by using the base of the next element to skip the definition of a segment length. Mini-SST entries encode permissions of large memory regions.

The MMP was designed for Mondrix, which is an extended version of the Linux OS [WRA05]. It aims to provide an isolation of software activities and kernel modules within the kernel space. Since kernel software activities, e.g. kernel threads, have full access to all system resources the system behavior is similar one of a system with a single address space. The MMP was implemented in a modified version of the Bochs x86 simulator as a hardware extension of a commodity x86 computer system [Law96].

3.2.4.5 Micro memory protection unit (UMPU)

Kumar et al. present a coarse grained micro memory protection unit (UMPU) for tiny embedded processors, called the Harbor framework [RSC+07]. The UMPU was implemented on an AVR ATmega103 8-bit MCU and enhances the implementation of function call and memory instructions of the MCU to perform run-time checks required for the enforcement of a software-fault isolation (see Section 3.3.1) [RKS07].

The UMPU implements a memory map checker to validate memory accesses made by a software module. It enforces a protection model that ensures that a software module can only write to memory sections associated to the module's domain. A domain is a distinct subset of the MCU's data memory. Each module resides in exactly one domain. The operating system uses a separated domain and has access to all other domains. On each memory write operation the memory map checker unit intercepts the physical access into the data memory. It stalls the processor execution to perform an address translation and a check of the permission bits in the memory map. In case of an access violation a memory map checker panic signal is asserted. But it is not entirely clear how this signal is handled. It can be assumed that the processor performs a system reset.

Cross domain calls are enabled by export functions. Each software module can offer a set of functions publicly accessible by other domains. A control flow controller ensures that control is never given out of a domain, except to exported functions. On each cross domain call, triggered by an enhanced call instruction, a cross domain call unit stores the calling domain ID, the stack bound and the return addresses on a separate *safe stack*. The safe stack resides in a protected region of data memory. It cannot be modified by a software component.

Furthermore, to provide control flow integrity within a domain, the UMPU copies the return address on each call instruction to the safe stack. The enhanced return instruction is modified to use the return address present on the safe stack.

3.2.4.6 Sancus

The authors of Sancus present a security architecture for networked embedded systems. The architecture supports remote software attestation without trusting any software on the device. The trusted computing base is completely implemented in hardware. The architecture enables mutually distrusting parties running their software on a sole node by isolating text and data sections [NAD+13].

The isolation is enforced by a hardware-based memory access control unit. The system's memory is divided in protected and unprotected non-overlapping sections. The memory access control logic of Sancus enforces that data in a protected section is accessible only while code of an associated text section is executed. To prevent ROP attacks, the text sections that have access to protected data can be entered by jumping to a well-defined entry point only. The total number of protected sections has a fixed upper bound, which can be configured when synthesizing the soft-core processor. The memory access control unit is instantiated for each protection section and is pure combinational, so that it needs no extra clock cycles.

To simplify the implementation of software modules the authors provide a compiler extension based on LLVM [LA04] and a support library that offers an application programming interface (API) to perform commonly used functions. The compiler extension processes annotated C source codes. The annotations are used to indicate functions and data of a protected section.

3.3 Software-based memory protection

Due to the lack of a hardware-based MPU in most off-the-self MCUs a feature set for a memory protection in TSSs can be inspired by the large range of software-based memory protection techniques proposed for commodity desktop/server systems. Although these software-based approaches are focused on safety they can be adapted for security demands.

3.3.1 Software-based fault isolation (SFI)

The software-based fault isolation (SFI) originally proposed by Wahbe et. al [WLAG93], is a fundamental technique for restricting the address ranges of *unsafe* operations. It is a GCC compiler approach that provides a sandboxing of application modules by re-writing the binary code so that a distrusted module cannot escape its protection domain. The approach is based on two basic techniques:

Segment matching is a software encapsulation technique, which inserts checking code before every unsafe operation. The checking code determines whether the unsafe instruction addresses the correct memory segment. If the check fails a trap is raised to handle the error outside. The segment matching technique requires four dedicated

registers[1]: two to hold the addresses of code and the data segment, one to hold the segment shift amount, and one to hold the segment identifier.

Address sandboxing can reduce the SFI overhead. It sets the upper bits of the target address to the correct segment identifier so that a distrusted module cannot produce an illegal address. Address sandboxing requires five dedicated registers: one to hold a segment mask, two to hold the code and data segment identifiers, and two to hold the code and the data segment.

An *unsafe instruction* is any instruction that uses an address that cannot be statically verified. Most control transfer instructions use a target address relative to the current program counter and store operations to static variables use an immediate addressing mode. These operations can be statically verified. However, jump and store operations through registers cannot be verified. These operations are wrapped by the SFI approach.

Since nodes of WSNs provide a single address space only, in its community SFI is a large research topic. For example, the t-kernel introduces a process called *naturalization* that patches the application code at load time so that all branch instructions are redirected to the t-kernel [GS06]. A more detailed introduction of t-kernel is given in Section 3.4.4.2. The Harbor framework aims to enforce similar goals: it forbids memory write operations and jumps to addresses outside the module's domain. In contrast to the original implementation of SFI, Harbor does not support a static partitioning of the available memory and uses a shared stack within all software activities. Therefore, protecting the shared stack is a design challenge of Harbor [RKS07, Ren07].

3.3.2 Control flow integrity (CFI)

The control flow integrity (CFI) enforcement ensures that the software execution follows a path determined by the control flow graph (CFG) created ahead of time [ABEL05a, ABEL05b]. CFI has many similarities with methods that attempt to discern program execution deviation from a prescribed static CFG [RCV+05, OSM02, VHM03]. These methods are primarily focused on fault-tolerance that concern a one-time random bit-flipping in program state or in registers. The CFI approach is able to hold an adversary back that is able to change data memory, e.g. by exploiting the program stack. It ensures that an attacker can never execute instructions outside the legal CFG. It does not aim to provide fault tolerance or unauthorized memory access.

The methods of Abadi et al. [ABEL05a] and Oh et al. [OSM02] are similar in the way how control flow is restricted. Both instrument the binary by labels and checks. In contrast to the work of Oh et al., which uses run-time checks that are evaluated at the destination of all branches and jumps, Abadi et al. propose run-time checks at the jump or branch source. Hence, CFI is able to prevent jumps into the middle of a function to bypass a security check. Since the technique of abnormal control flow modifications is an essential step in many exploits, the CFI enforcement is an effective instrument against a broad variety of common attacks. Furthermore, CFI is a general technique that is not focused on the prevention of buffer overflows and other vulnerabilities only [WK03].

[1] Dedicated registers are used by the inserted code only and are never modified by the distrusted code module.

XFI, software guards for system address spaces, is a comprehensive protection system that offers fine-grained memory access control and fundamental integrity guarantees for systems state [EAV+06]. The system was designed for Windows on x86 architectures to run modules safely within both kernel mode and user mode. Similar to SFI and CFI, XFI combines static analysis with inline software guards that perform checks at runtime. It provides control-flow as well as program-data integrity. Program-data integrity is ensured by memory-access constraints in which a memory access is either into the module's memory or into contiguous memory regions to which the supervisor system has explicitly granted access.

3.3.2.1 Stack protection

Many countermeasure techniques have been proposed to prevent control flow attacks based on bored buffers. On commodity systems these techniques are based on program and stack randomization, stack canaries or enforcing pages to be writable or executable. However, most of those countermeasures rely on hardware that is unavailable on TSSs. Software-based solutions suitable for TSSs are based on compiler modifications.

Cowan et al. uses stack canaries as illustrated in Figure 3.10. They present a compiler extension that places a canary word between local variables and the return address. When the function returns, it checks first that the canary word is unmodified before jumping to the address stored on the stack. The approach assumes that the return address cannot be manipulated without touching the canary word. Due to the fact that an attacker has to write sequentially into the stack it is very difficult to over-write the return address word without modifying the data close to it [CPM+98].

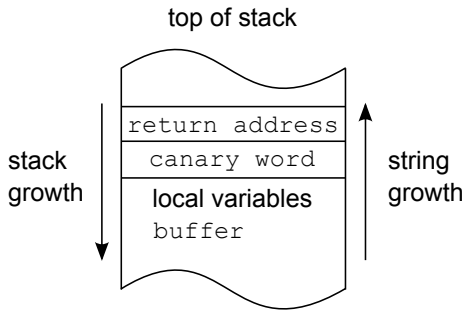

Fig. 3.10: StackGuard places a canary word next to the return address on the program stack to detect stack manipulations [CPM+98].

Alternatively a separation of the data stack and the control flow stack can be used to provide an isolation of control flow information from the regular data allocated on the stack. The StackShield approach copies the return address to a safe area and checks the return address before jumping to the address pointed by the address word [Ven00]. In a similar manner, Xu et al. [XKPI02], XFI [EAV+06], and instruction based memory access control (IBMAC) [FPC09] make use of two stacks: a regular data stack and an isolated control flow stack (see Figure 3.11). There are several possible layouts in which those two stacks could be arranged in the memory. The data stack should lie at its original position to provide maximized backward compatibility. Data allocation will work in exactly the same way as before and no modifications to the compiler are necessary.

Since double corruption attacks [Ale05] would allow an attacker to corrupt the data pointer first and then modify the control flow stack located anywhere in the memory a hardware-

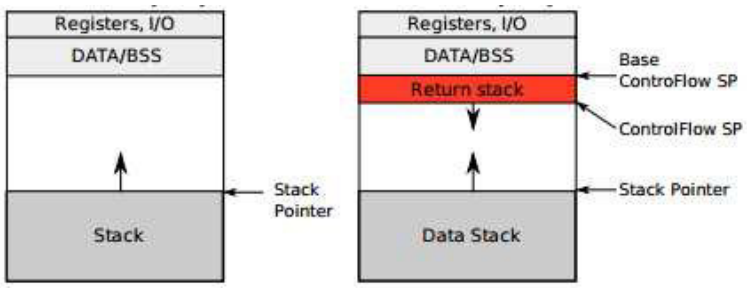

Fig. 3.11: Memory layout with a traditional single stack and with separated data and control flow stacks [FPC09].

based protection of the control flow stack is necessary. The authors of [FPC09] present a hardware extension for AVR-based MCUs that restricts the manipulation of the control stack to the `call` and `ret` instruction, so that a direct modification of the return address is not possible.

3.3.3 Safe languages

Approaches on the language level fundamentally build on properties of the programming language to achieve memory-safe code. It involves a compiler or a pre-processing tool in establishing memory safety. Furthermore, all safe languages, for example, Java [Gos95, GJSB00], Modula-3 [CDG+92], Safe-C [ABS94], and CCured [NCH+05] rely on garbage collection and use a number of run-time software checks before and after memory operations to maintain this property [DKAL05]. The safe languages approaches require that the source code must be available in the respective programming language. Retrofitting the programming language is most widely used in legacy code to achieve memory safety while requiring modest changes to existing code only [Sti12].

3.3.3.1 Java

Java started in 1991. The project's goal was to build a software environment for small distributed embedded systems [Gos95]. The environment, as illustrated in Figure 3.12, was designed to cope with heterogeneous networks and to build long-lived reliable systems. In particular, it was required to compile software that can be deployed in a network independent from the system's architecture, where it runs later on.

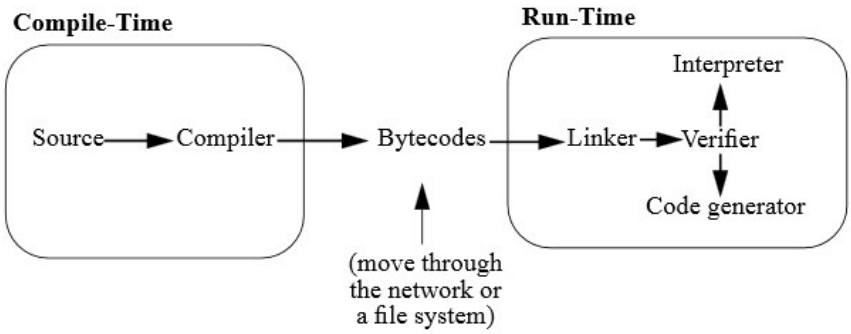

Fig. 3.12: The Java software environment [Gos95].

Java compiles the program sources to a byte coded machine independent instruction set. In contrast to common byte-codes it has an unusual amount of type information, restricts the use of the operand stack, and has a heavy reliance on symbolic references and on-the-fly code rewriting. Type information coded in the opcode, e.g. there are different load opcodes whose implementations are identical, except the type of data they load. Each entry of the program stack and each local variable has a type. An important property of Java is a static assignment of the current stack type by induction. Each instruction has the inductive property that it gives its type state before its execution, so that the final type state of a instruction execution can be determined before. Furthermore, in case of more than one execution path to a single point, it is ensured that the instruction finished with exactly the same type state. These restrictions have a number of important consequences. There are a number of properties that can be checked statically. The interpreter can be much faster due to the fact that pre-execution assumptions are possible. Furthermore, these properties improve the system's security: pointers can be treated as capabilities. Applications cannot forge them, they cannot get around them, and all the access restrictions are enforced. It ensures that a variable, which is defined private, cannot be accessed by a member of a class that does not own it.

The Java software environment includes a byte-code verifier that traverses the byte-code, constructs the type state information and verifies the types of the parameters to all the op-codes. The verifier acts as a kind of gatekeeper, which does not allow code to be executed that has not been verified before.

3.3.3.2 Cuckoo

Beside Java, memory- and thread-safe system services can be implemented by using Cuckoo. The language was primarily designed for the compilation of application-specific services of extensible OSs [WW05]. The language is both syntactically and semantically similar to C. Hence, legacy code can be easily translated and new programs may be written in a manner familiar to many developers.

The Cuckoo language is focused on the ability to ensure safe access to memory even when code is executed by multiple threads. It provides fine-grained control over memory usage and achieves run-time performance close to that of unmodified binaries. The language authors describe a program as memory safe if it fulfills the following two conditions:

- memory writes to nor read from any memory location that are not reserved for the programs by using a trusted run-time system are not possible and
- program jumps to any location that does not contain trusted code, which is either generated by a trusted compiler or accessed via a designated entry point to the trusted run-time system, are not permitted.

The memory safe conditions of Cuckoo are very similar to traditional memory safety definitions, except for their dependence on a trusted run-time system. In particular, Cuckoo claims that a correct compiler and a trusted run-time system produces only memory safe programs. In contrast to Java, which achieves also memory safety for multi-threaded system, it does not require a large virtual machine (VM) and its corresponding garbage collection. Instead, Cuckoo has many similarities to SFI, it integrates also run-time checks, but does not provide partial memory safety only.

3.3.3.3 Program transformation systems

Most of all embedded systems are implemented using the C programming language. But this language encourages programming at the edge of safety. It makes programs efficient but also vulnerable to safety and security violations, as introduced in Section 2.1.1. But the usage of safe languages is not a realistic solution for everyone. Hence, various approaches are introduced to extend unsafe languages such as C. The goal of these retrofitting approaches is to design a language that has the safety guarantee of safe languages while keeping the syntax, types, semantics, and idioms of the original language intact.

All these approaches share the concept of combining static analysis on source code by more or less complex run-time checks on pointer operations. The run-time checks are inserted in the compiler output. The output can be either a programming language file that has to be further processed by a native compiler or directly machine code instructions.

Safe-C provides a complete error coverage enabled by a simple set of program transformations. To enforce access protection Safe-C extends the notion of a pointer value to include information about the referent. The idea is similar to tagged pointers used in many Lisp implementations. The program transformation includes three basic operations: pointer conversation, check insertation, and operator conversation [ABS94].

Cyclone is a safe dialect of C with an acceptable adaption effort for legacy C code. It was designed to prevent C programs from buffer overflows, format string attacks, and memory management errors. It differentiates between regular pointer, `never-null` pointer, and `fat` pointers. It is forbidden to use pointer arithmetic on regular pointers. Furthermore, on each access pointers are checked against null. `never-null` pointer must never hold a null value, which is always checked when the pointer is initialized. `fat` pointers are three-word pointers that include boundary information. This is the only type that may be used in pointer arithmetic. To avoid dangling pointers the `free()` function is disabled and replaced by a garbage collector [JMG+02].

CCured is a program transformation system that adds type safety guarantees to existing C programs. It categorizes pointers into four different kinds: safe-pointers, seq- (three-word fat) pointers, wild-pointers, and rtti-pointers. Safe- and seq-pointers are similar to Cyclone regular and fat pointers. Wild pointers are fat pointers that include a type tag, as size tag, and information that allows determining for each word whether it is a pointer or not. Dereferencing a wild pointer includes a type check and a bound check. Furthermore, on each memory write operation the tag data needs to be updated, which makes wild pointers very expensive. To reduce the number of wild pointers, a form of physical subtypes is supported. The rtti-pointers tracks the actual type of the pointed object. This type of pointer can be used in upcast and checked downcast operations [NCH+05].

Deputy is a C compiler that is capable to prevent common programming errors [CHA+07, Con15]. It includes out-of-bound memory checks and many other common type-safety errors. Deputy is based on simple program code annotations that describe pointer bounds and other important program invariants. Similar to the never-null pointer of Cyclone, the expression of

invariants at interfaces allows that pointer checks can be propagated back to the caller. Code, compiled with Deputy, can be linked directly with code compiled by other C compilers. But Deputy does not include checks for memory deallocation and dangling pointers. Due to its small run-time overhead, Deputy is used in Linux device drivers and also in safe variants of sensor node operating systems, as Safe TinyOS and Safe Contiki (see Section 3.4.4.2).

3.3.4 Binary instrumentation

Program transformation systems and common type-safe languages, as Java or Modula-3, are used on the source code level. All these approaches can be used only if the program is available and written in the specified language. Investigations on binary programs or exotic languages are possible with binary instrumentation tools. Popular memory error detectors based on binary instrumentation are Purify [HJ92], Dr. Memory [BZ11], BoundsChecker [Bor15], and Discover [Ora11]. The most common system is Valgrind [NS07].

Valgrind is a dynamic binary instrumentation (DBI) framework, which makes it easy to debug, profile and detect dynamic error in binary programs [NS07]. It supports the x86, AMD64, x390x, ARM and PPC platforms and runs on Linux, Android and Mac OS X as well as experimentally on FreeBSD and NetBSD. Valgrind uses an just-in-time compiler to translate a binary program into a temporary, easy to use, platform-independent byte-code, so called Vex IR. Afterwards, the Vex IR byte-code can be processed by Valgrind tools. Finally, the processed byte-code is compiled into native machine code and can be executed directly on the target host. Additional instrumentation can be used to check the shadow memory on application's load and store operations [SN05].

AddressSanitizer is able to detect errors within the stack memory and global variables [SBPV12]. It consists of two parts: an instrumentation module and a run-time library. But in contrast to retrofitting unsafe languages, AddressSanitizer works at the very end of the LLVM optimization pipeline and does not require any source code modifications. Nevertheless, due to library requirements, AddressSanitizer can be used with C or C++ programs only.

3.3.4.1 Hardware-based memory error detection

The detection and correction of internal memory errors is possible by using error-correcting code (ECC) memories. The SafeMem approach utilizes the ECC bits to detect memory leaks and some classes of memory errors [QLZ05]. A hardware-programmable state machine residing in a memory that associates each memory byte with a state and treats each access to the memory as an event is implemented by MemTracker [VRSP07]. But both SafeMem and MemTracker fail to detect attacks that allow arbitrary memory writes.

Arora et al. [ARRJ06] propose an architectural extension to the Xtensa processor to reduce the performance penalty of the CCured approach. The proposed solution replaces memory violation detection operations with custom instructions, which are executed on a co-processor. The SafeProc approach extents the ISA of a SimpleScalar simulator [BA97] to include safety instructions that provide compile time information on pointers and objects [GGD+09]. A highly compatible and complete spatial memory safety for C programming

language is given by the SoftBound approach [NZMZ09]. HardBound is a hardware implementation of the SoftBound approach and aims to achieve a spatial memory safety on the C programming language as well [DBMZ08].

An additional approach for a hardware-based memory error detection is AHEMS [TLKI14]. The AHEMS framework consists of two parts: a hardware extension for run-time checking of memory safety and a source-code instrumentation for connecting software and hardware. The hardware extension was integrated in the Leon3 processor. In contrast to other approaches AHEMS uses an asynchronous security engine that is connected by a FIFO to the processor core, so that an attack can only be detected after the fact. To enable run-time notification of the hardware about memory events the program code is instrumented, by using the CIL source-to-source compilation [NMRW02].

3.3.5 Virtualization

The virtualization technology was originally introduced in the 1960s, as a method for dividing the system resources of mainframe computers between different applications [PG74]. Hardware virtualization or platform virtualization refers to the creation of virtual machines, so called guest machine, that acts as a real computer system. As illustrated in Figure 3.13, all software components, including the OS, on virtual machines are decoupled by a virtual machine monitor (VMM) or hypervisor from the underlying hardware. Common computer systems are idle for plenty of their time, so that sharing the physical resources between multiple OSs can save costs, power, cooling and floor space. All these aspects make virtualization an attractive choice for many vendors.

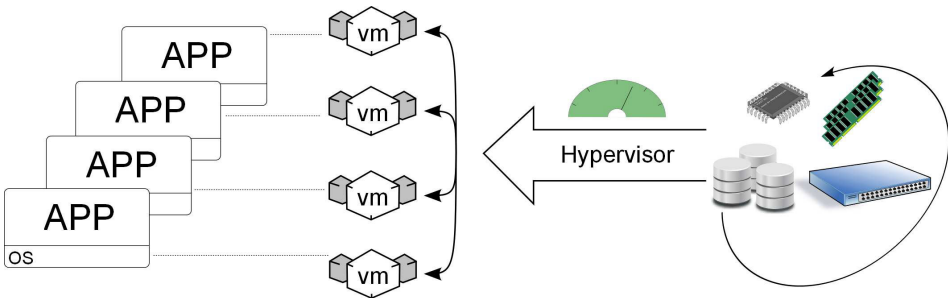

Fig. 3.13: Hardware virtualization (VMWARE).

In the last recent years virtualization was used in many research and commercial environments to run multiple operation systems concurrently on a single physical platform [SK10]. Because of the technical perspectives of virtualization for a secure isolation of software instances on a single system, security aspects have become a hot topic. In contrast to traditional single instance systems the security of each hosted instance can be increased by a virtualization layer without any hardware extensions. In addition, old systems can be enclosed by a modern secure platform, which opens a continuous use in current environments. Especially embedded systems with long living applications benefits from these technologies.

Based on the implementation of the VMM four different types of virtualization can be identified. A classification of these types is shown in Figure 3.14. This classification differentiates in a first step between emulation and native virtualization. The emulation makes an execution of a hypothetic machine or different host and guest machines possible, payed by a significant drawback in performance. The native virtualization allows an execution of parts of the guest

system on the host processor without any overhead, but requires a secure and reliable mechanism to yield the host CPU back to the VMM. Native virtualization is further differentiated into fully- and para-virtualized systems. On some hardware platforms full virtualization is very difficult[2]. These systems can benefit from a para-virtualized setup in which parts of the guest system are adapted for their virtualization.

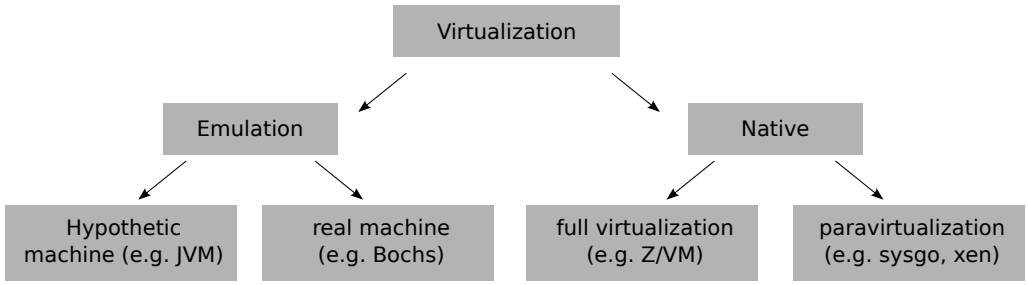

Fig. 3.14: Classification of virtualization schemes [NDB10].

In the following we will briefly introduce virtualization techniques. We will sketch few implementations of VMMs that are representatives of the introduced classes. Examples of hypothetic machine were already given in Section 3.3.3 by the Java VM, Valgrind, and LLVM [Gos95, NS07, LA04].

3.3.5.1 Instruction set emulation

The instruction set emulation virtualizes a system including the processor's ISA. The instruction set emulator reads the binary instructions of the guest system and carries out the instructions that contain data. It operates in a similar way as a real processor accessing real data. An example listing of an instruction set emulation is given by Listing 3.1. A typical emulator implements a very large switch-case-instruction where each case implements a single instruction.

Listing 3.1: Instruction emulation.

```
while (executing) {
    switch(RAM[PC]) {  //Grab the opcode at the program counter
        case ADC: //Add with carry
            A = X + RAM[PC+1] + CARRY_FLAG(FLAGS);
            PC += ADC_SIZE;
            break;
        // <snip long list of cases>

        default: //Invalid opcode!
    }
}
```

Listing 3.1 illustrates the major drawback of an instruction set emulator. It needs various instructions to emulate a single binary instruction. But in the context of security, instruction emulation strictly decouples the guest system from real resource. The emulator has access to all micro-operations of a single instruction. Beside security, software debugging benefits

[2]The x86 architecture implements instructions that behave different in the super-visor and the user-mode. These instructions cannot be virtualized and must be executed by the trap-and-emulate method.

significantly from the fine-grained execution and makes error detection much more comfortable.

Instruction emulators are available for a broad variety of real ISA. A very well-known example is the **Bochs** emulator with a history of 25 years. It is a highly portable open source IA-32 PC emulator written purely in C++ running on most popular platforms [Law96]. It includes emulation of the CPU, common IO and a custom BIOS. Beside Bochs, **QEMU** is an emulation core of a virtual machine provided as free software by Bellard [Bel05]. The core idea of QEMU is to emulate μ-operations of the target CPU. The μ-operations are generated offline by an additional tool, so called `dyngen`, and are used as a replacement of emulated instructions. To reduce the number of μ-operations a static set of registers, instead of all possible combinations, is used for the different operations. At run-time the emulation core analyzes a code block and translates it by using the micro operation in a native executable code block. The translated code blocks are cached in a translation cache. A new code block ends at a jump or at an instruction that modifies the CPU state. The translated code blocks are chained by using the program counter and CPU state information as input of a hash table. Bellard states that his system is 30 times faster than Bochs and 1.2 times faster than Valgrind.

In the context of embedded systems instruction emulators are available to run TSS applications on PC systems, e.g. the **Avrora** instruction set emulator for the ATmega MCU [TLP05]. The **MSPsim** is a Java-based instruction set emulator for the MSP430 MCUs [EDF+08]. We will introduce the MSPsim in detail in Section 7.2.2.1.

3.3.5.2 Native virtualization

Native virtualization is an established technology in common desktop and server systems that has been pushed by the VMware company in the last fifteen years. It is mainly used for server consolidation and desktop virtualization. In the first decade of this century PC processor manufacturers have introduced hardware support for x86 virtualization. Beside the x86 mainstream processor virtualization is long-time established for mainframe processors.

The virtualization is enabled by the hypervisor, an additional software layer between host hardware and virtualized guest. Popek et al. classified two types of hypervisors: a bare-metal hypervisor and a hosted hypervisor [PG74]. An illustration of these two types is given in Figure 3.15.

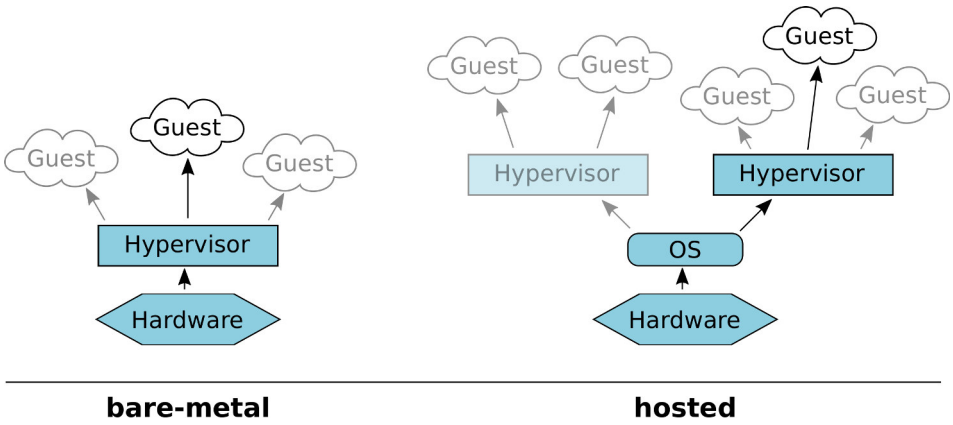

Fig. 3.15: Classification of hypervisors. Bare-metal hypervisors are running directly on the host's hardware and a native OS is not required. Hosted virtualization requires an OS instead.

Type-1: a bare-metal hypervisor runs directly on the host's hardware. Although the hypervisor controls the hardware, it does not implement drivers for all components. Resources can be delegated to a guest, which runs as a process on the host and implements the needed hardware driver.

Type-2: a hosted hypervisor runs as a process of a host operating system and uses its hardware abstraction. The hypervisor abstracts the host OS so that guest and host ISA may differ or have not to have any dependencies.

A bare-metal hypervisor is offered by VMware ESX, z/VM, Xen, and **XtratuM**. XtratuM is a hypervisor specially designed for embedded real-time systems that supports the x86 architecture but also the SPARC-based LEON architecture[3]. The XtratuM hypervisor can be used to build an MLS system [MRC10].

3.3.5.3 Para-virtualization

Para-virtualization is a virtualization technique that presents an interface to the guest that is similar, but not identical to that of the underlying hardware. It requires modification on the guest system to support to the hardware interface. The modified interface reduces the portion of operations within the guest that are difficult to execute in a virtualized environment. A successful para-virtualization system may allow the hypervisor to be simpler and reduces the performance degradation inside the guest.

The term "Para-virtualization" was initially used by Whitaker et al. with the Denail lightweight VM [WSG02]. The technique is also used in the context of various modern hypervisors, as Xen, VWware, L4, and XtratuM. The **Xen** project provides a hypervisor based on μ-kernel primitives. It allows the execution of multiple OSs on a single system [BDF+03]. But due to the para-virtualization approach, conventional OSs, which are not para-virtualization-aware cannot run on top of Xen. Therefore, Xen and other para-virtualization systems are rather compatible with open source OSs, as Linux, FreeBSD, NetBSD, OpenSolaris et alii. Nevertheless, other systems are supported by providing a kit of para-virtualization-aware device drivers. Steinberg at al. presented **NOVA**, an OS virtualization architecture focused on constructing a secure and efficient virtualization environment with a small trusted computing base (TCB) [SK10]. It is based on a μ-hypervisor inspired by the L4 μ-kernel and the para-virtualized OS L4Linux [HHL+97]. We will introduce the L4 μ-kernel in Section 3.4.2.

3.3.5.4 Virtualization on sensor nodes

Hardware virtualization in embedded systems was introduced by [Hei08]. Due to the lack of memory protection schemes, a native virtualization cannot be implemented. Nevertheless, virtualization on TSSs without an MPU became possible by using instruction set emulation. The instruction set emulation may be based on a hypothetical or on a real machine. While a real machine emulation uses the same ISA for the guest as well as the host system, the hypothetical machine emulation does not require identical ISAs.

Weerasinghe et al. propose a lightweight module isolation for sensor nodes. It is based on a virtual instruction set close to the 'generic', underlying physical ISA but augmented

[3]The LEON is a soft-core processor developed by Gaisler Research for embedded control tasks [Gai03]

with special 'emulated' instructions for memory management and calling across protection domains [WC08]. The concept does not impose a real VM, instead memory management instructions are identified in a pre-deployment stage and replaced by virtual instructions that are emulated are at run-time. Due to the pre-deployment transformation process the number of emulated instructions is quite small in comparison to 'real' virtualization. Nevertheless, real virtualization is already implemented on TSS. In the following we give a brief overview about well-known approaches.

SPUMONE

SPUMONE is a lightweight CPU virtualization layer for embedded systems [KYK+08]. The authors motivate the need of a hybrid operation system, which consists of a real time OS and a standard OS. The presented approach is compared with WOMBAT and RTLinux. All these approaches are paravirtualized systems, but SPUMONE requires a minimum of modifications in the guest OS.

The system was implemented for the SH7780 CPU. On the CPU runs the SPUMONE layer and on top of it a Linux guest and a μITRON RTOS guest. Both OSs were modified to be executable on the SPUMONE layer. The needed modifications have been minimal. To catch the real time requirements, the RTOS gets an higher priority than Linux. Interrupts for Linux will be delayed until the real time tasks are finished. The execution of Linux is possible in idle phase of RTOS only. Measurements have shown that the SPUMONE layer's penalty is tolerable and it has a small effect on cyclic real-time applications. The guest OSs run natively on the CPU. The virtualization layer intervenes the execution in interrupt handler only. A guest OS is running until an interrupt occurs.

Security is not covered by the presented approach. The address spaces of guest OSs are not isolated. Each task can modify data of another guest as well as data of the virtualization layers. The guest kernel is running with high privileges and has full system access. To enforce a secure virtualization it is also important that a guest may not modify the interrupt table. The SPUMONE layer cannot detect any memory modifications and in the worst case guest OS never returns to the virtualization layer.

Maté/Bombilla

Maté presented by Levis et al. is a VM for TinyOS [LC02]. It is implemented as an additional component on top of TinyOS and allows the execution of small programs received via the wireless network interface. The VM's memory footprint consists of a few bytes only, which makes an integration on small sensor nodes such as on the family of Berkeley motes possible. The instruction set architecture (ISA) of Maté is limited to a small number of one-byte instructions, which can be deployed in 24-bytes capsules. Programs larger than one capsule can be spread over a multiple number of capsules with few limitations. However, the ISA is capable to handle typical sensor node tasks. It includes three different types of instructions: basic instructions, message manipulation instructions, and stack and branch instructions. Maté has three execution contexts corresponding to the three events: timers, message reception, and message send. Each context has two stacks, an operand stack and a return address stack. The size of both stacks is limited to sixteen and eight entries, which has been determined by practical lessons.

SwissQM

SwissQM is a VM that uses a small subset of the integer and control instructions of the Java virtual machine (JVM) specification [MAK07]. It includes 59 instructions, whereof 37 are identical to the JVM specification. Similar to Maté, two fixed-size stacks are used and the interpreter is running on top of TinyOS. But SwissQM is more efficient and mature. However, these types of VMs are focused on typical sensor node sequences as *sampling*, *processing*, and *sending* or *sampling*, *processing*, *merging received data*, and *sending*.

Java-capable VM

The usage of Java on embedded systems started with the Squawk system [SSB03]. The Squawk implementation was primarily focused on a mature and small JVM. As a target platform smart card devices with a low power 32-bit processor and few k-bytes of random access memory (RAM) have been addressed. Beside a small footprint the JVM was as much as possible written in the Java language to simplify portability and debugging. This approach is already evaluated by the dynamically extensible virtual machine (DVM) [BHR+06], the JVM TakaTuka [ASE+08], and the Darjeeling [BLC09]. KESO is a multi-JVM system for deeply embedded systems [TSWSP10]. In can be used on top of Autosar to run multiple SW-C on a single MCU [WSS11].

All these Java-capable VMs are not focused on security. Especially, Maté that allows user-defined instructions to access data outside the VM, offers a large vector for possible attacks. Security seems to be out of scope of VM research in the area of WSNs.

3.4 Modern operating system architectures

In the previous sections we have introduced access control and memory separation schemes in general. These techniques are usually tightly coupled with operating systems. In particular, commodity desktop and server systems use OSs to implement general hardware abstractions. In recent years OSs on TSSs became more and more common and must be considered when implementing new software on these systems.

The history of operating systems starts with the mainframe computers in 1960s. The earliest computer systems have been used without any operating system, jobs have been hand-made on punched paper cards and scheduled by human system operators. The genesis of modern operating systems came with libraries of support code, which would be linked to the user's program to assist its operations. In the following, more and more optional software features became standard in every OS. This has led to the perception of an OS as a complete user system including utilities, daemons, and a user interface (shell or window system).

In the following, we will focus on the OS kernels, which include services in the more restricted sense of operating systems only. Especially within the context of deeply embedded system that do not feature a user interface or user applications, the operating system is mainly the system kernel including all types of device drivers. Operating system kernels can be differentiated in three architectures: monolithic kernels, microkernels, and exokernels. Next, we will introduce these three architectures in detail.

3.4.1 Monolithic kernels

The monolithic kernel architecture implements the entire operating system in the kernel space. The kernel is the sole component that is executed in the highest privileged mode. It includes basic primitives as process management, memory management and inter-process communication (IPC). All device drivers, the network stack, as well as file systems are part of the system kernel. In modern OSs kernel components can be loaded on demand. Nevertheless, loadable components are integrated in the system kernel with the same privileges. A monolithic operating system kernel with loadable components will also be called a hybrid kernel. But from the perspective of security it is more similar to a monolithic kernel.

Most desktop or server operating systems, such as Microsoft Windows, Linux, or BSD, are monolithic or hybrid kernels. The main reason for using monolithic kernels is their outstanding performance. Implementing all system services in a single address space reduces the number of processes and IPC, which is seen as major performance problem of fine-grained architectures. But, due to the growing complexity of system kernels, they were more prone to software bugs and became increasingly difficult to maintain. Similar to the growing system kernels user space applications in large systems with 64-bit address space and huge amount of memory ask for a more fine-grained isolation of software activities [KCE92].

3.4.2 Microkernels

Microkernel, μ-kernel, systems were built long before the term itself was introduced, e.g. the system nucleus by Brinch Hansen [Han70] or the Hydra system by Wulf et al. [WCC$^+$74] were developed long before. The first-generation of μ-kernels, like Mach [ABB$^+$86], still provide multiple services within the kernel space. In 1995 the L4 μ-kernel was introduced by Jochen Liedtke [Lie95b]. The L4 μ-kernel is based on the L1, L2, and L3 μ-kernel, which have been developed by Jochen Liedtke in previous works. In contrast to the previous μ-kernel versions L4 and QNX [Hil92] are second generation μ-kernels that have a dramatically reduced interface and improved IPC performance. Liedtke proposed that the inefficiency of a μ-kernel is not a problem of the idea itself. Moreover, it is a problem of overloading of functionality or improper implementations. Hence, the proposed L4 μ-kernel is reduced to provide only three basic primitives: *address space*, *threads and inter-process communication (IPC)*, and *unique identifiers*. Further services must be implemented outside the kernel in user threads.

Due to the basic concepts address spaces and IPC of the L4 μ-kernel are also central paradigms for building a secure platform on TSSs, the following descriptions are focused on the L4 μ-kernel.

3.4.2.1 Address spaces

The basic idea of L4 is to support a simple address space construction scheme, which can be used outside the kernel. The μ-kernel has to hide the underlying hardware implementation and has to provide basic operations to manage address spaces only. There is an address space σ_0, which represents the physical memory as well as memory mapped resources. Further address spaces are constructed and maintained by user processes on top of the σ_0 address spaces. For this purpose the μ-kernel provides three operations:

- **Grant.** An owner of an address space can grant any of its pages to another address space, if it is already accessible to itself. Hereupon, the granted page is removed from its address space.

- **Map.** Similar to the grant operation, an owner of an address space can map any of its pages to another address space. But, the mapped page is accessible in both address spaces.

- **Flush.** The owner of a page can flush any of its pages. Through this the page is removed from all address spaces, which had received the page directly or indirectly by a grant or a map operation. Hereupon, the page is accessible by the flusher only. Therefore, a user must accept a potential flush, when they receive a page from another address space.

The grant operation is needed in very special situations only. In general, the operation is used when a page should be passed through. An example of a grant operation is illustrated in Figure 3.16. The process F combines the two underlying address spaces f_1 and f_2 into one unified address space. The process F grants the page to the process $user\ A$. By granting the page instead of mapping, the page is removed from its address space and must not be maintained any longer. The resulting mapping, denoted by the thin line, exists between f_1 and $user\ A$ only. But by modifying the access rights during the grant operation the process F can restrict the access on the page without bookkeeping it later.

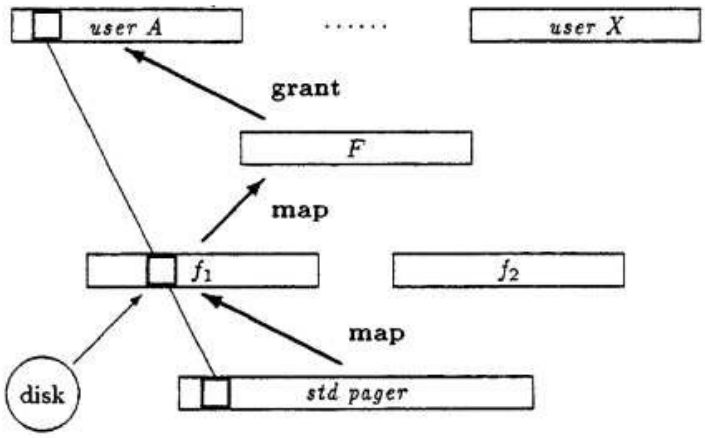

Fig. 3.16: Example of address space mapping and granting [Lie95b].

3.4.2.2 Inter-process communication (IPC)

IPC by message passing is one of the central paradigms of most distributed systems and applications. Thread communication via IPC is one fundamental feature of the L4 μ-kernel. L4 IPC is always synchronous, so that sender and receiver must negotiate a rendezvous time to communicate. If a communication peer is not ready the other one is blocking until a timeout occurs or the peer enters the communication. L4 supports three different types of IPC: short-IPC, long-IPC, and flexpages.

Short-IPC transfers data in processor registers. It requires no extra copy operations and can be executed with the lowest IPC cost. But the amount of data is limited to the processor's register number and size.

Long-IPC , also called direct IPC, sends messages up to two megabytes of data. This message type needs copy operations and the kernel might establish a temporary mapping [Lie93].

Flexpages are pages mapped or granted from the sender to the receiver. It can be used to transfer a large amount of data without copy operations.

Beside data transfer, IPC is used in L4 for synchronization, wakeup-calls, pager invocation, exception handling and interrupt handling. By sending messages from the μ-kernel to user threads functionality can be implemented in user-space instead of in the privileged kernel space, which increases system's security and dependability in a significant manner.

In address spaces used by a multiple number of threads the IPC performance can be further improved by local-IPC [LW01]. A local-IPC is a short IPC operation between threads in a single address space. The whole operation can be handled inside the address space and requires only a storage of the program counter and the program stack pointer. A switch to the μ-kernel is not necessary, the thread can be switched directly instead.

3.4.3 Exokernels

The exokernel is an operating system from the Massachusetts Institute of Technology (MIT) [EKO95]. In contrast to traditional OSs, exokernel provides as few as possible hardware abstractions to application developers. Applications can communicate with the hardware much more directly. The exokernel is very tiny, since its functionality is limited to ensuring protection and multiplexing of resources. It is simpler than μ-kernels and monolithic kernels.

The MIT exokernel manages hardware resources as processor, memory, disk storage, and networking. The kernel represents the processor as a timeline. A program can allocate intervals of time and can yield the rest of its time to another program. Processor events, as interrupts, exceptions, and the begin and end of a time slice, are delivered to programs by the kernel. Furthermore, the physical memory pages are allocated to programs, where access is controlled by capabilities. A program can send a capability to another program to share the page.

Exokernel-like technologies have coined multiple terms: nanokernel, picokernel, cache kernel, virtualizing kernel. But most of these are variants of each other. For example the term "nanokernel" was introduced by Bomberger et al. in 1992 and describes the kernel of KeyKOS. KeyKOS is a small capability-based system designed to provide security sufficient to support mutually antagonistic users [BFH$^+$92]. Nevertheless, KeyKOS belongs to the family of microkernels, so the terms are used analogously.

3.4.4 Operating systems in TSSs

Operating systems of TSSs must consider the MCU with their very limited processing power, memory and battery life-time. The hardware of TSS mostly does not provide a memory protection scheme and different privilege levels. Hence, a classification as presented in Section 3.4 is not possible. Furthermore, we can differentiate between OSs that are linked to a single application or systems that feature a system kernel and loadable modules.

The number of OSs for TSSs and their feature set is really widespread. In the following, the basic design philosophies of real-time OSs and security features of some example OSs for TSSs are introduced.

3.4.4.1 Design philosophies

The design of TSS OSs is close to the design of real-time OSs. Their major design goal is not high throughput, but rather a predictable time to accept and complete a task. Key factors of these systems are minimal interrupt latency and minimal thread switching latency considering the limited resources of the underlying hardware. We can differentiate two design philosophies for the OSs of TSSs:

Event-driven OSs follow the programming paradigm of the event-driven programming in which the flow of the program is determined by an event. Event-driven OSs force a programmer to structure and program an application as a state machine in terms of tasks and event handlers. An activity starts with an event, mostly an interrupt, and runs until handling of the event is finished.

Time-sharing OSs implement a preemptive or co-operative multitasking. System activities, e.g. threads, fibers, or coroutines, share the processor by using time slices. Preemptive multi-tasking requires a central component that allocates time slices to activities and revokes the processor when a time slice is consumed. In a cooperative system each component has to yield the processor to other processes.

Examples for event-driven sensor node OSs are TinyOS [LMP+04], SOS [HKS+05], and Reflex [WKN08]. These systems use mainly a component-based architecture, where a component consists of attributes and methods. Some of them are only internally used and some of them are public accessible. Event-driven OSs can be implemented with a sole stack, which reduces memory consumption and simplifies task handling. Tasks are non-preemptive and are scheduled by the OS. The scheduling algorithm may be very simple, like FIFOs (TinyOS), or more complex and configurable, as implemented by Reflex.

Although time-sharing systems are more vulnerable to program errors that cause the whole system to hang, their design is much simpler, e.g. tasks do not need to be reentrant. Time sharing sensor node OSs are for example Contiki [DGV04], MANTIS [BCD+05], RETOS [CCJ+07], and RIOT [BHG+13]. These systems use a periodic timer interrupt that triggers an event. But the periodic event increases the energy consumption. Hence, alternative techniques as a variable-tick time rate are investigated for battery-driven systems. In this type of systems context switching and scheduling operations are the major source of overhead.

The event-driven paradigm may be favored for the resource constrained environments. The thread-based model of OSs simplifies the application's implementation and will be easier to program for conventional programmers [CCJ+07].

3.4.4.2 Security in OSs of deeply embedded systems

Although embedded systems are very common in various application areas, their implementation is mostly based on raw C implementations or sometimes bare assembler. With the last fifteen years research on resource constraint devices, the design of operating systems was intensified. Within this period OSs with very different justifications were developed. But security was mostly out of scope of these systems. In the area of automotive systems the OSEK/VDX has pushed a standardization of OSs and services. The work continued in the automotive open system architecture (AUTOSAR) project with the AUTOSAR OS. AUTOSAR provides a well-defined system architecture for a broad variety of hardware platforms. But its architecture is focused on safety. Security was upcoming within the last few years.

In the following we will present the security features of a few OSs for wireless sensor networks and deeply embedded systems. This overview does not cover the broad variety of OSs in this area, but provides a comprehensive overview of research work focused on security.

TinyOS

TinyOS is the de facto standard OS of WSNs [LMP+04]. This OS consists of a rich collection of components ranging from low-level parts to application-level logic. All components are written is nesC, which is a dialect of the programing language C [GLvB+03]. The OS does not distinguish between kernel and user components and a memory protection is not included. The nesC compiler can detect some interrupt concurrency bugs and permits function pointers. The final system image is statically linked, which facilitating resource usage analysis and code optimization such as in-lining.

Although memory protection is not part of TinyOS, compiler-enforced safety is a standard mechanism of TinyOS, e.g. Safe TinyOS [CAE+07] uses Deputy [CHA+07] to make TinyOS and its applications type-safe, preventing pointer bugs from cascading into memory corruption and random consequences. On a safety violation, Safe TinyOS reboots the entire node. The Neutron's compiler and run-time extension of TinyOS minimizes the safety violation cost by introducing micro reboots that efficiently recover from memory safety bugs [CGK+09].

We can state that TinyOS security is focused on network protocol security [KSW04, MWS04, ZSJ06, WAR06]. Memory protection schemes within the context of security are to the best of our knowledge not available.

Contiki

Contiki is an open source OS developed by Dunkels at the Swedish Institute of Computer Science (SICS) [DGV04]. The OS is completely written in C. The system is separated in a fixed kernel and loadable modules. The kernel includes the program loader, basic services, and a small set of libraries. The programming model of Contiki is based on *protothreads*. A protothread is a programming abstraction that features event-driven and time-sharing philosophy. An protothread is invoked by the kernel in response to an external or an internal event.

Contiki supports pre-emptive threads as part of its multi-thread (MT) library. Each MT thread has its private stack segment and program counter that are saved during program switches. But threads share their address space with all other threads. An isolation among threads is not implemented.

Similar to the Safe TinyOS, Paul et al. presented safe Contiki OS [PK09]. The approach modifies the Contiki build chain to integrate Deputy. Furthermore, the Contiki core has been adapted to annotate each pointer access. The primary goal of this approach was Safety. A secure isolation of software components is out of its scope.

SOS

SOS is a dynamic OS for sensor nodes [HKS+05]. The authors of SOS assert that it brings dynamic and general-purpose OS semantics without significant energy and performance sacrifices to a sensor network OS. Like TinyOS, SOS is an event-driven OS that uses a component module design. The OS consists of a statically compiled kernel and dynamic loadable binary modules. Modules are linked with the kernel on load-time. Each module provides a set of functions, which it uses and offers. Modules can communicate with other modules and the kernel through message passing, wherefore each module implements a message handler. Messages, send by a module, are annotated with the destination's identity and stored in a kernel FIFO. The kernel invokes the module's handler function, which was registered at module's load time, to deliver the message. Execution control is transferred back to the kernel when the handler terminates.

SOS provides a dynamic memory allocation scheme. Dynamic memory is used to store the module's state and to create messages. The kernel tracks ownership of the allocated memory. Ownership can be transferred to pass data easily through different modules. But the write operations of a module are out of control of the SOS kernel. An enforcement of protection domains was introduced by the Harbor extension of SOS [RKS07]. Harbor ensures that modules can write only to the protection domain that they own. Although the approach lacks the support of shared protection domain, SOS supports message passing, which can be used to implement cross-domain communication.

CiAO

The CiAO operating system is an academic research OS developed at the University of Erlangen-Nuremberg [LHSP+09]. The design of the operating system is focused on configurability and extensibility by applying aspect-oriented programming (AOP) principles, based on AspectC++ [YKC06]. CiAO implements large parts of the AUTOSAR OS API, including memory and timing protection.

The primary development platform for CiAO is the Infineon TriCore, a 32-bit architecture mostly used in automotive industry (see Section 3.2.4). CiAO memory protection can be applied in four different degrees: no protection, kernel protection, application protection, and task protection. Protection domains are defined for the kernel, applications, and even task-local data depending on the used degree. But CiAO protects the data and not the code, it traps only when the code modifies protected data [LSH+07].

According to the application model of AUTOSAR OS, CiAO supports trusted and non-trusted applications, which form the protection realms. The communication between applications

is implemented by synchronous remote procedure calls (RPCs), *trusted functions* and *non-trusted functions*. A trusted function is a function exported by a trusted application to be called by another application. It is executed in a privileged, trusted context. In contrast, a non-trusted function is a function that is executed in an none-trusted application in an unprivileged protection context. It can be called by other applications as well.

CiAO is used as the underlying operating system of KESO. It was adapted to support the CiAO AUTOSAR implementation and to integrate the application model. Furthermore, it supports the MPU-based memory protection scheme.

t-Kernel

The t-kernel approach aims to overcome the lack of hardware support for privileged execution and address translation by performing extensive code modifications at load time [GS06].

After initializing its own working environment, the t-kernel loads the applications from external storage. The application's code is split in small blocks of consecutive instructions, so called code page. When the control flow of an application reaches a new code page, the page is loaded from the external storage and modified in a way that the modified application runs in a collaborative manner. The modification process is called *naturalization*.

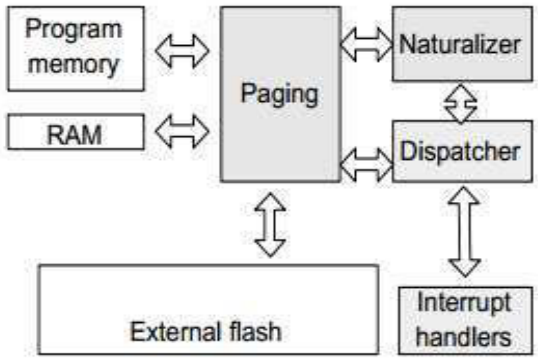

Fig. 3.17: Structure of t-kernel [GS06].

Naturalization is the key component of t-kernel. It ensures that the application yields the CPU to the kernel frequently. Therefore, the naturalizer modifies all branching instructions to load the jump destination address in dedicated registers and inserts jumps to the dispatcher instead. The dispatcher checks the destination to validate that the corresponding code page is already loaded. If it is not available, the page is loaded and naturalized. Afterwards, the program counter is redirected to the demanded entry point of the code page.

Based on naturalization the t-kernel provides virtual memory space larger than the physical data memory of the underlying hardware. Heap memory can be swapped out. Furthermore, the naturalizer ensures that the application stack cannot invade the kernel stack, the return address cannot be misused to jump outside the naturalized program, and the heap access is always valid.

FreeRTOS

FreeRTOS is a popular open source real-time OS based on a μ-kernel architecture. The

OS was ported to more then 30 platforms [Bar10]. It provides extended thread priorities and memory allocation schemes and is a very compact system with an outstanding execution performance. Similar to Contiki, FreeRTOS supports coroutines, which implement very simple and lightweight tasks with very limited use of stack, which fits perfectly to the requirements of TSS.

In complement to FreeRTOS SafeRTOS was developed by *WITTENSTEIN* high integrity systems. Both OSs share the same scheduling algorithm and have similar APIs. But SafeRTOS is completely written in the C programming language to meet requirements for certification according to IEC61508[4] at a safety integrity level (SIL) of 3, which is the highest possible certification for software-only components [Bar07]. SafeRTOS is included in the ROM of some TI Stellaris MCUs. This allows the usage of the operating system without having to purchase a source code license.

[4] IEC 61508 is an international standard for functional safety of electrical/electronic/programmable electonic safety-related systems.

CHAPTER 4

Security enhanced tiny scale systems

This chapter introduces the concept of a platform for the implementation of security enhanced TSSs. Where safety related approaches as presented in the previous section are focused on the prevention of any faulty or accidental memory access, in the context of security a secure isolation must ensure that a certain SA can access only those resources that are assigned to its protection domain. In addition, the approach must consider a malicious access that is performed with intention, whether clever or unclever to bypass security mechanisms.

The presented approach of a secure isolation on security enhanced TSSs is based on four basic principles:

- tailor-made **data spaces** to implement protection domains,
- enforcement of software **flow integrity**,
- **trustworthy instance** to control any access on any resources, and
- fine-grained definition of **access control**.

The enforcement of a secure isolation requires that a TSS application is separated in multiple SAs that run in their own tailor-made data spaces. Considering the characteristics of deeply embedded, event-driven systems such a modification must be done at function level. The presented approach conserves the original program flow and makes a fine-grained access control between functional units possible. The implementation of complex security schemes becomes possible by an ahead of time application analysis. Especially due to availability of the program sources compile-time analysis can be used for further optimizations.

The isolation of data spaces is a key property that can be provided only by a layer that decouples the SAs from the underlying hardware. As illustrated in Figure 4.1, we will call this trustworthy instance the security nucleus (SN). Security can be guaranteed when any access to any resource is controlled by the trustworthy instance. Therefore, each SA, except the non-isolated SAs, is extended by a nucleus gate to interact with the security nucleus. Non-isolated SAs are necessary to set-up system primitives and are executed within a secure boot-strap.

In traditional OSs, a trustworthy instance as our SN is implemented by a combination of a higher privileged software instance and a memory protection mechanism provided by the underlying hardware such as an MMU. Instead of an MMU we propose a memory protection tailor-made for the limited resources of TSSs. Furthermore, where safety related approaches control write operations and function calls only, in the context of security all operations must be taken into account. We will expound that similarities to safety related approaches can be used. In addition, we will discuss concepts for a SN implemented as a hardware unit or as a pure software solution. In particular, a software solution with high security demands may be based on a VM monitor that emulates an ISA. In case of lower security demands

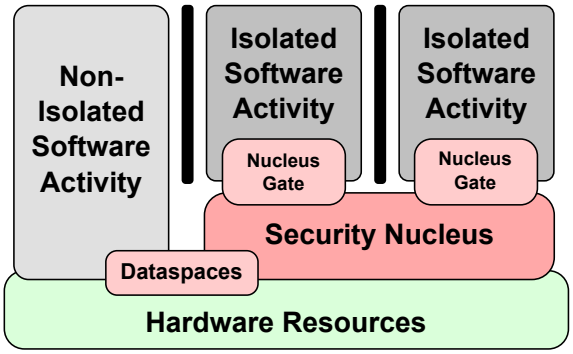

Fig. 4.1: Isolation of software activities in TSSs. A trustworthy instance, the security nucleus has to decouple software activities from their underlying hardware.

an approach based on SFI techniques may be sufficient. A detailed discussion of these approaches is essential for a secure isolation with system's performance in mind and will be given in this section.

An isolation of software activities on function level requires that a fine-grained description of activities, components, and operations of TSS applications and their permissions is possible. Therefore, we will present an adaptation of the RBAC model to provide such a fine-grained description and access control model. Since the software of TSSs is mostly available in source code a compile-time preparation approach is possible. The presented system is a compile- and run-time co-design that uses a preparation step at compile-time to reduce run-time overhead as much as possible.

In the following subsections we will present the conceptual approach of the four basic properties. Although the concepts should be independent from their implementation, it is essential to discuss the conceptual it in combination with few implementation details constraint by TSSs. In depth discussion of implementation details is given in the following chapters.

4.1 Tailor-made data spaces

In legacy operating systems the enforcement of protection domains is provided by a virtual-memory framework controlled by the memory management component of the system kernel. The mechanism is based on hardware features provided by the processor's MMU. A broad variety of implementations has been presented in Section 3.

The authors of [ALE+01] present the concept of the data space[1] paradigm in the context of the L4 μ-kernel system. They defined the terms data space and region. Due to similarities between μ-kernels and our approach for security enhanced TSSs, we introduced both terms as the basic elements for the resource management on TSS. But both terms need an adaptation to be applicable to TSSs. Hence, we define the terms as follows:

Data space A data space is an unstructured data container, which contains any type of data. In TSSs data spaces contain non-volatile memory, RAM and memory mapped IO resources such as peripheral registers.

[1] The term data space was coined by Beyer et al. [BDJ88].

Region A region is a view on a data space assigned to a software activity. Any access by an SA results in an access of a region associated with a data space.

An example of the concept is given by Figure 4.2. The software activity sa_1 has access on three data spaces associated with regions r_0, r_1, and r_3 to it. The regions are attached to the data spaces ds_0, da_1, and ds_3. The data space ds_3 is additionally assigned to a region r_4 of sa_2. ds_3 is a shared data space that can be used by both software activities.

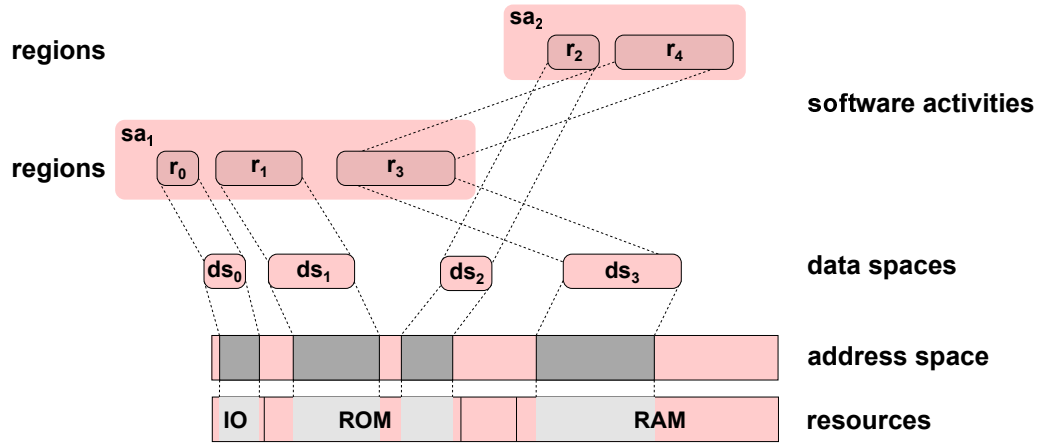

Fig. 4.2: Relationship between software activities, data spaces, regions, and address space.

According to the concept of the memory access control matrix (MACM) a TSS will have a set of data spaces D, which contains address space segments, and a set of software activities S. Furthermore, there is a set of permissions P, which specifies the type of access that is allowed for an SA to process data of a data space. A permission is expressed by the tuple $p(s, d)$, where $s \in S$, $d \in D$ and $p(s, d) \subseteq P$.

4.1.1 Data space descriptor

According to the needs of TSSs a segment-based memory management was chosen to describe data spaces. Due to its fixed-size granularity a page-based memory management would not be a reasonable choice. A paging with a page size of few bytes, which is necessary to meet these requirements, causes a large overhead and becomes very inefficient. Segments can be defined with a flexible size and a variable start address. Especially peripheral units may require a byte-granular segmentation to isolate peripheral registers.

4.1.1.1 Data space descriptor table (DDT)

In order to provide a systematic way to control access, an access matrix, a capability list, or an ACL can be implemented. The usage of a $n \times m$ access control matrix, where n is the number of processes and m is the number of objects is very inefficient. Especially TSS applications would have a very sparsely used access matrix. In case of using a capability list each process has to hold its capabilities. Since all resources have to be accessible by each process and a hardware-based implementation of dynamic lists is not possible, the resulting

data structure will be degenerated. The list would have nearly the same size as an access matrix.

Therefore, we propose an implementation of an array of access matrix elements. Each element includes the addressed object, expressed by the segment boundaries, and a capability field. The capability field includes the software activity identifier (SAID) and the permissions of the SA that gets access to the object. We assume that in TSS applications the number of private objects is much higher than the number of shared objects, so that the number of matrix elements, which addresses the same object, is quite small.

We store the access matrix elements in a DDT. On each memory access a DDT entry look-up is necessary. The look-up function inputs are the memory address that is accessed and the identifier of the SA that performs the access. It returns the access rights of the DDT entry if an entry exist, otherwise an empty set.

4.1.1.2 Data space boundary description strategies

When implementing segmentation on TSSs the description of a single segment becomes crucial. A single segment is bounded by a start and an end address. The description strategy of these boundaries has a significant impact on performance and overhead. It was initially investigated by H. Menzel [Men10] in his Master's thesis supervised by me. We analyzed four different strategies to describe segment boundaries. Usually a segment start is described by a physical address $D.start$ and the bit-width of the address must be equal to the address space bit-width. Other description strategies for the segment start might be possible but will always be a trade-off between complexity and performance. Since a minimal run-time complexity is crucial and can be enforced only by avoiding any transformations, we decided to use a physical address.

For the description of the segment size we can differentiate between the following four strategies:

End address (EA) uses an additional address $D.end$ to store the address of the segment.

$$x \in D \mid x \geq D.start \wedge x \leq D.end.$$

Size in bytes (SIB) uses an additional field $D.size$ to store the number of bytes of the segment. To be able to describe a single segment that includes the complete address spaces the size field must have the same bit-with as the address space.

$$x \in D \mid x \geq D.start \wedge x < (D.start + D.size).$$

Size in 2^n (SIT) uses an additional field $D.order$ to store the number of bytes of the segment described by an order of two. Here, the size can be described very compressed and the test is very simple, but the description is not as flexible as the size in bytes.

$$x \in D \mid (x \,\&\, \neg((1 \ll D.order) - 1)) = D.start.$$

Next address (NA) uses the start address of the next segment $D_{next}.start$ to indicate the end of the current segment.

$$x \in D \mid x \geq D.start \wedge x < D_{next}.start.$$

A comparison of the four strategies is given in Table 4.1. We compared the required memory, the look-up operations, possible segment overlapping, and the granularity of each strategy.

Table 4.1: Comparison of strategies to describe segment boundaries.

Strategy	Size	Look-up effort	Overlap	Fully addressable
EA	$2 \times addressWidth$	$2 \times compare$	yes	yes
SIB	$2 \times addressWidth$	$addition + 2 \times compare$	yes	yes
SIT	$addressWidth + sizeof(n)$	$shift + mask + equal$ $+addition + inverse$	yes	no
NA	$addressWidth$	$2 \times compare$	no	yes

The decision for using one of these strategies depends on two factors: the location of the DDT and the complexity of a DDT entry look-up. In commodity systems the page table or the segment table is stored in the main memory. In this case the size of the DDT is less important and the size of a sole DDT is less critical. In TSSs with their limited memory resources an external storage in a dedicated memory is an option. In such a system the size of the DDT might be even more critical.

4.1.1.3 DDT look-up engine

The DDT entry look-up time has a significant impact on the overall system performance. It includes the DDT entry look-up based on an address match and a comparison of the SAID. To check any memory access the operation must be performed on each memory access. In software as well as in hardware several search algorithms are available. On average, a look-up in a RAM consumes the time required to scan and compare one-half of the table. A significant better performance can be achieved by using a content-addressable memory (CAM) [Cor63]. In a high speed CAM, a look-up can be done in a single clock cycle [HW96].

In the following we will present two alternative approaches for a DDT look-up engine: a CAM-based design and a cache-based design. Both designs have advantages and drawbacks, which have to be taken into account when implementing tailor-made data spaces.

Content addressable memory (CAM)

A look-up unit using a CAM can be implemented as illustrated in Figure 4.3. A CAM as proposed by Helwig at al. [HW96] works with the *compare data* and a *mask* and performs a similarity operation inside the CAM array. When using the SIT strategy, the compare data is equal to the address accessed by the SA. But instead of providing the mask, each CAM array entry contains the data space size as an internal bit mask. Due to the fact that each data space can have an individual size the mask cannot be given as an external input. On a

look-up, the CAM array executes a parallel search over all entries and rises a hit line in case of a match. The hit line is translated in a DDT index, which is used to read the corresponding DDT entry. Depending on the organization of the DDT, the complete operation can be done in a single clock cycle. But the implementation requires an additional hardware unit and is not suitable for software-based approaches. Furthermore, the size of the CAM array is fix and cannot be adapted dynamically.

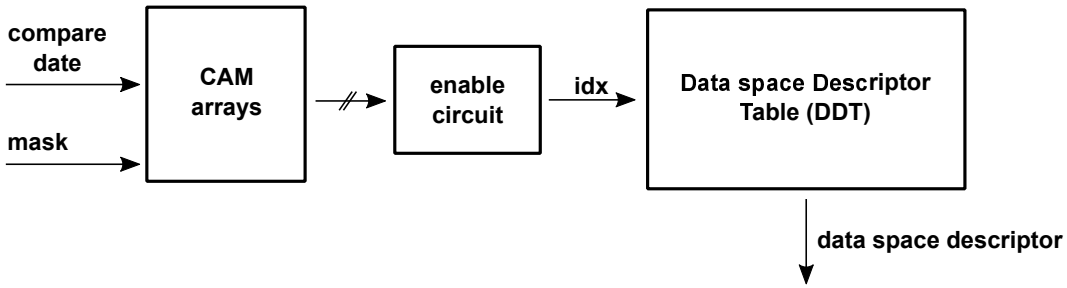

Fig. 4.3: Block diagram of a DDT look-up unit using a high speed CAM array element.

Since the CAM performs a parallel look-up on each element, in case of multiple matches multiple hit lines are raised. When coding the data space boundaries in the CAM array the enable circuit must be able to translate it in a single index or to perform a sequential DDT access on each index. In Section 5.1.2.6 we illustrate how multiple hits can be handled.

Data space lookaside buffer (DLB)

An alternative DDT look-up scheme can be implemented by using a data space lookaside buffer (DLB). A DLB, similar to a TLB, is a special cache that keeps track of recently used entries. The DLB contains the DDT entry that has been most recently used. The key to improve the look-up performance is to rely on the locality of the program code. When a look-up of a DDT entry is used, it will be probably needed again in the next memory access because the program flow of an SA has both temporal and spatial locality. A minimal DLB-based DDT look-up engine can be implemented if the number of DLB entries is one. As security policies should differentiate between data spaces containing data and program code, a sole entry is not suitable. Hence, at least two DLB entries are required. One entry for the data space that is used within the last instruction fetch and a second entry for the last data access. Instead of using a DLB with two entries, both can be implemented in separate DLBs so that a costly matching can be avoided. As shown in Figure 4.4, a multiplexer that is controlled by the fetch state flag can be used to select the current DLB. The same signal is used to control the update unit.

Since each DLB contains a sole entry the match unit can support more complex operations. Therefore, the implementation of all data space description strategies becomes possible. Furthermore, a software-based implementation will also benefit from a DLB. A DDT search is only started in case of a DLB miss. The main drawback of the DLB-based DDT look-up engine is its limitation to a first match. It is strictly demanded that the DDT entries are not overlapping, otherwise the look-up might return a wrong entry.

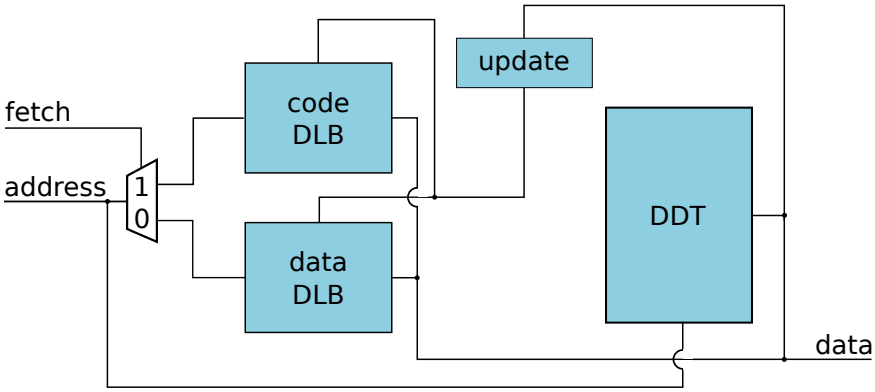

Fig. 4.4: Block diagram of a DDT look-up unit using a DLB.

4.1.2 Shared data spaces

An isolation of SAs on TSSs requires a shared memory. Data must be exchanged between SAs for communication as well as for data processing. In the master thesis of H. Menzel [Men10] and previous publications related to this work [SLM11, SLM13], the concept of group segments (data spaces) has been introduced. While a private segment is assigned to only one SA, a group segment can be used by a multiple of SAs. Figure 4.5 illustrates the management of group information presented in Stecklina et al. [SLM13]. The extended information, in particular the *activity mask*, is stored in a separated table, the group lookup table (GLT). The activity mask contains a bit for each SA, which indicates that the corresponding SA has access to the shared segment. The permissions are stored in the segment lookup table (SLT) entry [2], so that all SAs of the GLT entry get the same permissions. Hence, individual permissions for different SAs on a data space are not supported by that concept, so that an extended approach is needed.

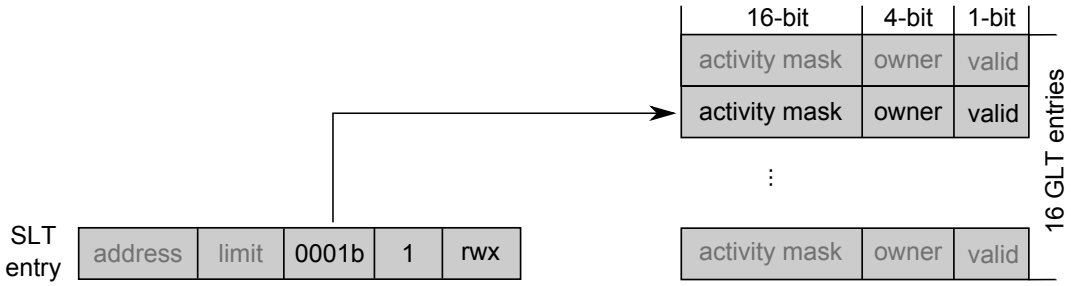

Fig. 4.5: Group look-up table GLT with extended information to implement shared segments. The size of the GLT is limited to the size of the owner field of the SLT entry [SLM13].

Investigations on applications for TSSs have shown that *shared memory* is either

- a method for inter-process communication (IPC) or
- a method for conserving memory.

IPC by shared memory is a very fast way to communicate on a single machine. Concurrent processes can exchange data by writing into the shared memory area. In addition, shared

[2] The approach presented in [Men10, SLM13] uses the term SLT instead of DDT. We use SLT here to avoid any confusion with the previous publications.

memory is used to conserve memory by direct access to data, which otherwise would be copied. Especially event-driven systems benefit from the concept of delegates. Whereas, IPC based on shared memories is mainly used in preemptive and task-based systems, TSSs are focused on event-driven systems, i.e. delegation is key.

In the context of secure systems it is essential that delegation of data spaces is trustworthy on the sender's and receiver's side, otherwise a *non-trustworthy SA* can use delegation operations to impose a malicious data space on a foreign SA. Therefore, it is very important to provide these operations with care. We will further discuss the security aspects of data space delegation in Section 7.1.1.3.

4.1.2.1 Granted data spaces

The security platform supports delegation by the concept of *granted regions*. In contrast to the approach of a GLT we simplified the permissions in a way that we define that an owner has full access to the data space that it owns. Furthermore, we extended each data space descriptor by a capability field. As illustrated in Figure 4.6, the capability field includes the SAID of an additional SA that gets the permission P, defined in the *permissions field*. The permissions field includes the *grant* capability, which gives an SA identified by $SAID$ the permission to grant the data space to another software activity $SAID^*$. On each grant operation the permissions P^* granted to another SA must be a subset of the original permissions P. Hence, $P^* \subseteq P$ is always true. If an SA drops the grant capability the SA $SAID^*$ cannot grant the region to any other SA $SAID^{**}$.

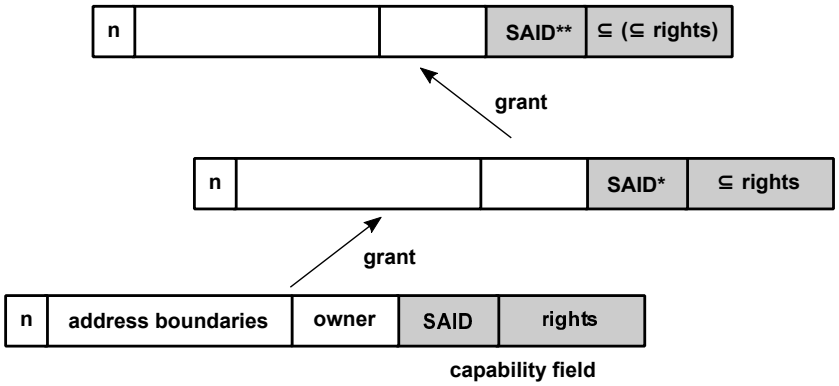

Fig. 4.6: The grant operation allows the delegation of data spaces to foreign SAs. On the grant operation only the capabilities field, including software activity ID $SAID$ and permission mask P, of a DDT entry n are changed.

Since the data space descriptor includes only a single capability field on each grant operation the granting SA, except the data space owner, loses its permissions on the data space. The data space is delegated from one SA to another SA. Therefore, an SA has permissions on the data space for a limited time only. A concurrent access by SAs that do not own the data space is not possible. Granted data spaces prevent a shared access of SAs, which do not own a data space, so that a bypass of the data space owner is not possible. Furthermore, the concept of granted data spaces reduce the number of active data spaces at run-time. Especially on TSSs the number of data spaces may become a significant factor and may be limited by the available resources.

4.1.2.2 Mapped data spaces

The grant operation allows us to delegate a data space to a foreign SA. On each grant operation the permissions can be reduced but the boundaries of the data space stay unmodified. We can construct scenarios in which granting of a whole data space is unwanted. Therefore, we added an additional operation that allows the creation of a subregion R^* of a data space D or a region R. We call this operation *region mapping*.

The map operation allows a software activity to delegate a subregion of a data space to another software activity. On each map operation the data space boundaries as well as the capability field can be modified in a way that $R^* \subseteq \{D, R\}$ and $P^* \subseteq P$. The map operation requires a new DDT entry to store the new information, which makes a map operation more expensive than a grant operation.

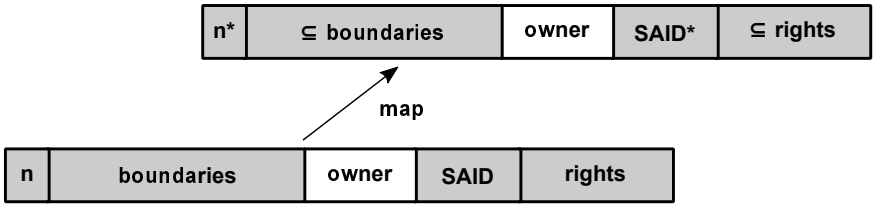

Fig. 4.7: The map operation allows the delegation of a subregion of a data space to another software activity. On the map operation the boundaries, the owner as well as the capabilities can be changed, which requires a new DDT entry.

A map operation requires that the mapped region is a subregion of the original one. To avoid that a mapped region can be expanded later by using the append operation, a link is required that indicates that a parent data space exists. Furthermore, the DDT entry look-up must be able to handle multiple regions of a single address. In Section 5.1.2 we will introduce implementations of hardware-based DDT look-up engines that are able to handle the problem.

4.1.3 Data space capabilities

Traditional OSs usually support three permissions on objects/files: *read* (r), *write* (w), and *execute* (x). But these permissions are not adequate to meet the requirements of secure systems. Within the context of TSSs, we identified six basic operations on data spaces: read, write, execute, resize, map and grant. We have mapped these operations onto seven capabilities. Here, we have split the operation resize into a shrink and an append capability. An overview of the seven capabilities is given in Table 4.2.

We have reduced the write capability to a modify capability. In contrast to the traditional write operation the modify capability does not allow any modification of the data space size. Instead we introduced two additional capabilities, which permit an SA to modify the data space or region size.

We did not specify a special capability to create a data space. Rather, an SA will have any capability to an unused data space. It can allocate a data space by mapping the data space to itself. Afterwards, it has exclusive access to the data space. We are aware that this immanent capability can be used to exhaust system memory. But the create operation is necessary to provide a kind of dynamic memory management. The remaining risk of exhausting system

Table 4.2: Basic capabilities of SAs on data spaces in TSSs.

Capability	Short	Description
read	r	data of the data space can be read
modify	w	data of the data space can be modified
append	a	the size of the data space can be expanded
shrink	s	the size of the data space can be reduced
execute	x	a software activity can execute data of the data space
map	m	a software activity can map the data space to another activity
grant	g	the software activity can grant the data space to another activity

resources can be limited by restricting the access on the MPU interface. We will introduce an example for such an implementation in Section 4.3.1.

Grant and map are the key operations to share data spaces between protection domains. We introduced the corresponding capabilities to have control over these operations. An SA can share a data space, which it does not own, only if it got the map or the grant capability. Hence, an SA retains control over data spaces that it has delegated to other domains.

4.2 Software activity flow integrity

A protection domain includes the dynamic data and the program code of an application. In a TSS with a fine-grained, secure isolation of software activities, a program that runs normally in a system with a single address space must be split in small program sections. Figure 4.8 illustrates the deployment of small program sections in different protection domains. We can say that the control flow of the program follows the protection domain in the same way as the data are located.

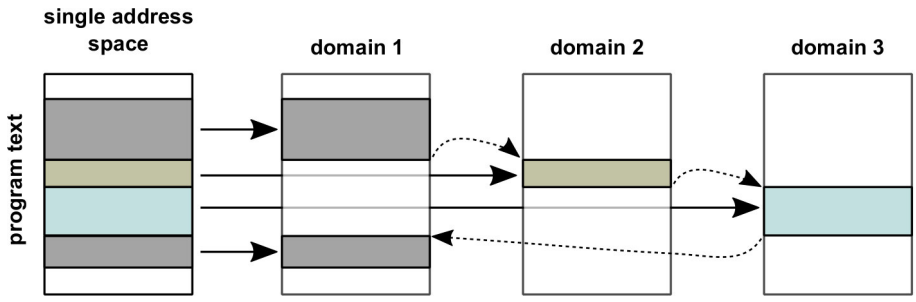

Fig. 4.8: The isolation of software activities forces the execution of program sections of single address space system in different protection domains.

In an ideal system the software flow continues in a new protection domain without any overhead, so that an instruction is executed in one address space and the next instruction is part of the new address space. Real systems require a more complex domain switch. At least it is necessary to save the old context and to enter the new domain at a well-defined entry point that redirects the call to the addressed function.

Commodity operating systems use IPC to invoke a remote function. Such an IPC is redirected to the system kernel, which activates the addressed process. Hence, each remote call

requires at least two context switches, which is a major bottleneck in fine-grained systems. Due to the limited resources of TSSs, any additional indirection will have a significant impact on the system's performance. Therefore, domain switches should be possible without any additional component, such as a kernel, or at least with a minimal overhead.

4.2.1 Cross-domain calls (CDC)

In a fine-grained TSS each SA is executed within its own protection domain. It has access to the data spaces that are associated to its protection domain. The software control flow of such a fine-grained TSS must follow the data that it needs to perform a required task. This means that a domain switch is necessary in all cases in which data of foreign address spaces are accessed. Since security is increased by a more fine-grained separation of SAs, domain switches become key. Especially, a domain switch must be possible at any program location to avoid restrictions in the software design. On the other hand domain switches are a dominant factor for the system's performance.

4.2.1.1 Domain switch

As an SA was defined as a single or a multitude of functions, a domain switch is placed at function calls. Although this is not a strict requirement it simplifies a well-defined isolation in a significant manner without any functional restrictions. Hence, a domain switch is similar to a remote procedure call (RPC) including marshalling, as described in Section 3.2.1.2. In TSSs we will call it a *cross-domain call (CDC)*. Using cross-domain calls at functions simplifies the parameter passing. In a well-structured software architecture a function should use parameters passed by the caller and private variables only. Therefore, the function's signature defines all parameters that have to be passed to the callee. Beside well-defined parameter passing, an access control at function level can be used to define operations that are allowed to be called by a foreign SA.

In a secure TSS a CDC will include the following operations:

- Caller saves the current software activity context.
- Caller marshalls the requested parameters.
- Caller clears all processor registers.
- Caller initiates the domain switch.
- Callee checks the requested operation.
- Callee restores its software activity context.
- Callee unmarshalls the parameters.
- Callee invokes the demanded operation.

Due to the limited resources of TSSs not all of these steps must be executed on each CDC. In particular, operations for parameter marshalling and unmarshalling may be very complex.

Therefore, parameters passed to a foreign domain must be carefully selected. Beside parameter passing, saving the context of the current SA is not necessary in each domain switch. Especially in event-driven systems a return of a cross-domain call may be unnecessary. An SA of such a systems handles an event and delegates further processing to another SA. Before delegation the SA can skip its context since its task is finished and control must never return. In case of a new event the SA starts with a fresh context. Any information regarding previously handled events can be neglected.

Each SA is identified by a unique ID. The domain switch must include an update of the current SAID. The update operation activates the new protection domain immediately. Therefore, it is important that the instruction just after the domain activation is executable by the new SA.

4.2.1.2 Parameter marshalling

We have introduced short-IPC, long-IPC, and flexpages with the L4 μ-kernel in Section 3.4.2.2. IPC is a key feature of distributed software systems. We use a similar approach for passing parameters by CDCs. Depending on the type and the number of parameters we must differentiate between two different types:

Short-CDC is a cross-domain call whose parameters can be passed by a small number of registers and the parameters do not refer memory objects.

Long-CDC is a CDC whose parameters refer to memory objects, so called *call by reference*, or whose number of parameters is too large to use registers.

Since an SA clears all processor registers, a short-CDC can be implemented without any additional operations. The parameter values are stored directly in the processor registers and are read by the callee after the domain switch. It is just required that both, caller and callee, use the same register conventions. A return value can be transferred by registers as well.

In case of a CDC with a large number of parameters or memory references a long-CDC must be used. In contrast to the term long-IPC used by the L4 μ-kernel we propose an approach more similar to the flexpages IPC. During a long-CDC, memory regions are granted to the callee. Copy operations are mostly avoided due to the lack of a kernel and the caused additional overhead. In particular, deep-copy operations that require following object references are highly critical. By using the grant operation, the parameters must be arranged so that a minimal region can be granted to the callee's address space. The arrangement can be done at compile time, which avoids run-time overhead, or at run-time by coping the parameters to a "transfer region", which is granted later.

4.2.2 Control flow checking

We introduced the concept of CDC to allow an SA to transfer control to another SA to access data of its data space. In such a system the control flow follows the data spaces that own the data that are needed to execute the requested task. Without any additional means, this

system will not provide a higher security level against intelligent attackers. An attacker can construct a control flow including CDCs to gather the needed information.

A higher security level can be provided by restricting the control flow such as presented by Oh et al. [OSM02] or Abadi et al. [ABEL05a]. At a domain switch, we can restrict the control flow to use certain functions of an SA to enter a protection domain. Hence, we follow the concept of UMPU [RSC+07] and CiAO [LSH+07] and group the functions of an SA into two types:

- **public** functions can be invoked directly by a foreign SA via a CDC and
- **private** functions can be used within a protection domain only.

Due to the fact that our concept implements a domain switch without any third instance, the enforcement of private functions can differ between software activities. For example, an SA providing a shared code section without private data that is used by various SAs can allow access to its functions. All functions will be public and a further access control is not necessary. Nevertheless, the isolation will complicate the construction of gatgets, as described in Section 2.1.1, and makes the overhead reasonable.

4.2.2.1 ACL-based CDC

Beside private and public functions, a more restrictive control flow checking becomes necessary if an SA provides public functions that access different private data. As shown in Figure 4.9, an SA that has more than one public function may offer access to a public function to a specific SA only. Such a restriction might be useful to protect functionality, e.g. F_3, or data, e.g. O_1, that can be accessed by the public function.

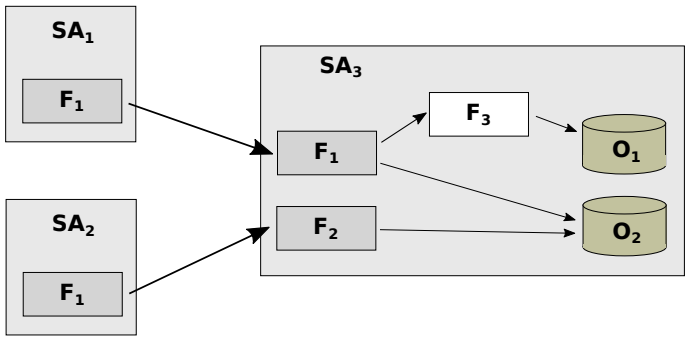

Fig. 4.9: ACL-based cross-domain calls. Function $SA_3.F_2$ can be accessed by $SA_2.F_1$, while $SA_3.F_1$ is public for SA_1 and SA_2. Function $SA_3.F_3$ is private and can be accessed directly by none of them.

A separation of functionality can be implemented by isolating each public function in an isolated protection domains, which requires additional protection domains and additional resources. Otherwise, we can construct scenarios in which a resource may be assigned to a protection domain and be used by a private and a public function, so that a separated protection cannot be implemented. Therefore, in case of multiple public functions, which can be accessed by multiple SAs, a more fine-grained access control is necessary.

We mentioned at the beginning of Section 4.2.1 a check of the demanded operation as an integral step of each CDC. Hence, we propose an ACL that assigns SAs to a function that

have access to it. Since access control is part of each CDCs, an ACL can be located within the protection domain of an SA with a minimal overhead. As an access control is coupled with an SA, the ACL might be optional depending on the application's security needs.

To avoid duplicated source code, the ACL functions can be located in a public SA that gets access to the ACL of an SA by the grant or the map operation. The management function can be public to each SA as well. Security critical data, e.g. the ACL, are mapped temporarily, so that they cannot be accessed by another SA in a non-preemptive system. In a preemptive system a monitor has to be implemented to make access control operations non-interruptible.

4.2.2.2 Program stack protection

An ACL-based CDC requires a trustworthy identification of the caller. In a message system a source ID is added by the communication channel. The μ-kernel approach also adds a source ID to each IPC message. We propose a similar approach by saving the caller's SAID in an isolated SAID stack controlled by the SN. On each update of the SAID the old value is pushed onto the SAID stack. It is mandatory that this operation has to be done automatically and may not be bypassed by an SA. For this reason, the SN must provide an interface to get the last SAID, which must be identical to the caller's ID.

A CDC is used as a replacement of a function call, in common software systems the caller expects that execution control is transferred back to it when the operation is finished. Such a return can be implemented by an additional CDC, but this can be used by a malicious SA to manipulate the return value. Figure 4.10 illustrates a function return that bypasses an intermediate function call. Hence, the SN must provide a dedicated return of aCDC that is based on the SAID stack.

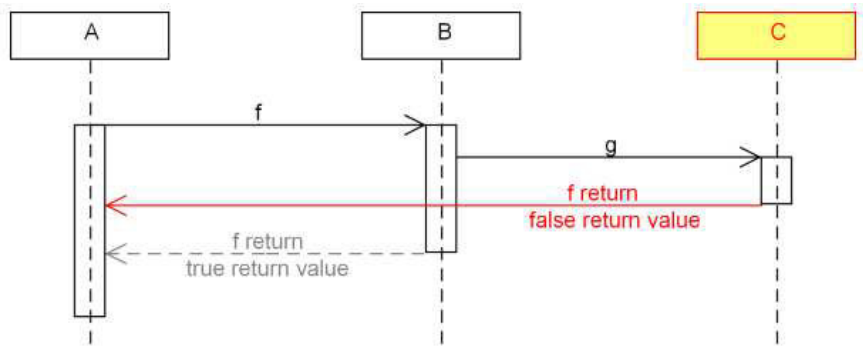

Fig. 4.10: The caller's SAID has to be stored outside the callee's protection domain otherwise a malicious software activity can manipulate the return path to bypass its caller [Ber12].

The SAID stack has to be stored in a protected memory area. Only the SN needs write access to this memory area. We will discuss the location of the SAID stack in Section 4.3.1. The stack can be implemented with an ordinary stack growing downwards. An implementation tailor-made for TSSs is possible by using a rotating buffer, as illustrated in Figure 4.11. The stack is implemented as a ring buffer and push (call) and pop (return) are implemented by rotate right and rotate left.

Using an ordinary stack limits the number of CDCs to the stack size. Any function call that breaks the stack boundaries will corrupt data near the stack or is hopefully blocked by the memory protection unit. A rotating stack will never corrupt the heap memory. In case of a

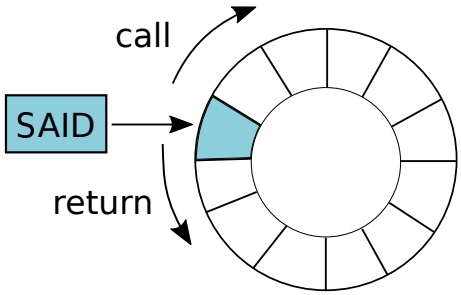

Fig. 4.11: A rotating window can be used to implement the SAID stack to store SAIDs on CDCs.

stack-breaking call graph, the return path is limited to the ring buffer size. In TSS an overwrite of return addresses on the rotating stack, may be uncritical in case of a return might be not longer used, e.g. for initialization functions. Furthermore, a rotating buffer allows us to implement a control flow without return. We already mentioned that such a flow might be suitable on event-driven systems. The processing of an event often starts with an interrupt and runs through various functions until the event is fully processed. Afterwards, the control flow returns to its initial state. The return path is unnecessary and can be omitted. We will illustrate such a system in more detail in Section 8.3.1, as this is out of scope of this thesis but might be a valuable part of further work.

4.3 Security nucleus

The term *system nucleus* was defined originally by Per B. Hansen [Han70]. Similar to a μ-kernel, the system nucleus supports the basic primitives address space management, IPC and unique identifiers. In the context of TSSs we define the term *security nucleus*. The main difference between system nucleus and security nucleus stems from the different characteristics of TSSs and our focus on security. We define the basic primitives *data spaces management* and *cross-domain communication* that substitute the primitives address space management and IPC slightly different:

Data space management includes the grant, map, and flush operation of memory regions between software activities. Since TSSs do not support virtual memory data spaces are mapped without address translation. All SAs share the same addresses of a single address space.

Cross domain communication includes domain switching and remote function calls. The data exchange is empowered by marshalling and data space granting. Cross domain communication is a key feature in security enhanced TSSs to isolate SA in real applications.

Due to the non-preemptive nature of TSS software, threads are mostly *coroutines* as defined by Conway [Con63]. Therefore, the security nucleus does not include an explicit thread management. Threads are substituted by SAs, which are managed at compile time and set up during systems boot-strap only, see Section 4.4.3. We propose that a security nucleus for TSSs must include at least the following components:

- DDT management,
- SAID management, and
- access control enforcement.

In the following we will analyze implementation concepts for a security nucleus in TSSs. The enforcement of a secure isolation can be provided either by virtualization techniques or by a hardware-based MPU. Virtualization is a pure software solution and can be used on off-the-shelf MCUs. The application of a hardware-based MPU requires an additional hardware component that must be integrated in the system architecture. It is mostly suitable in soft-core processors, which are implemented on reconfigurable hardware or application-specific instruction-set processors (ASIPs).

4.3.1 Hardware-based activity isolation

Due to the complexity of the security nucleus an implementation in hardware would be very resource-hungry. Therefore, only run-time critical operations are implemented in hardware. Complex management operations will still be implemented in software to keep the hardware size as small as possible.

The enforcement of a memory access control is a basic feature of a hardware-based MPU. For this purpose, the MPU needs at least access to the current SAID, the demanded operation, and the DDT. These requirements determine the integration, the memory, and the interfaces of the MPU.

4.3.1.1 MPU integration

For an enforcement of a secure isolation it is important that in case of an access violation the demanded operation is aborted and the violation is signaled to the processing core. Therefore, as illustrated in Figure 4.12, the MPU must be placed between the memory resources, including peripherals and special function registers (SFRs), and the processing core. In such a configuration any access operation can be easily aborted by controlling the chip select lines, which are used to enable peripherals, including memories and SFRs. The access violation can be signaled by an interrupt line. In addition, the configuration allows an MPU implementation similar to a peripheral unit.

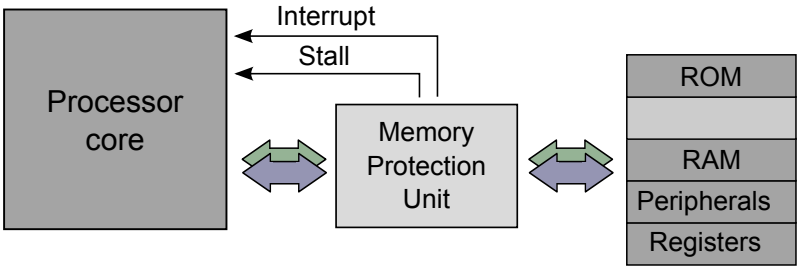

Fig. 4.12: MPU integration as an additional peripheral unit placed logically between the processor core and the memories and peripheral units. Access violations are signaled via a dedicated interrupt line and a stall line that is used to stall the processor core during more complex operations.

The demanded operation and the addressed data are set by the processor core. Read and write operations can be easily differentiated by the write enable signal. But an identification of an execute operation is more complex. It can be done by identifying the addressed memory[3] or by an additional signal. It depends on the processors architecture, we will describe this in Section 5.1.2 in more detail.

As shown in Figure 4.12, we need an additional line that stalls the processor core during complex MPU operations. Common MCUs have a direct memory interface in which the processor core expects that data are delivered within a defined period, usually within the next clock cycle. But for complex operations, e.g. on a DDT-entry look-up, a memory access may need more than the expected clock cycles and the processor core must be stalled until the MPU operation is finished. The stall signal can be used to control the clock tree of the processor core. Since the processor core is driven by a single clock source, it can be stalled safely at certain stages by stalling the clock source.

4.3.1.2 MPU memories

The MPU needs access to the ID of the current SA, the SAID stack, and the DDT. We can identify two different implementation variants: *external storage* and *internal storage*.

External storage

Powerful MCUs with an integrated MMU hold the segment table or the page table in the main memory. A page/segment table base address register holds the table base address of the current process. The MMU must provide the table base address register only. But the MMU has to share access to the main memory with the processor core, so that the memory bus might become a bottleneck. The access to the main memory can be minimized by using the DLB, as described in Section 4.1.1.3. Nevertheless, hardware specific look-up mechanisms as a CAM look-up cannot be implemented. Similar to the location of the DDT, the SAID stack can be stored internally or externally. In case of using the main memory the base address must be hold inside the MPU. A dedicated SAID register is not necessary. An SAID update can be identified by monitoring the memory address bus (MAB) instead.

Internal storage

Holding the DDT and the SAID stack inside the MPU makes the implementation of hardware-optimized look-up technologies possible. But the size of the DDT and the SAIDs are limited by the memory donated by the hardware designer. It is not possible to tailor the size of any of these memories to the application needs. We have already implemented the concept of an internal storage for an MSP430 MCU [Men10, SLM13]. We limited the number of segments to 128 and the size of the SAID to 4 bit. Both may be valid parameters for TSSs but require a memory of nearly 1 kB. In comparison to ultra low power MCUs, which include only few kilobytes of RAM, such an MPU design induces a notable overhead.

[3] Some MCUs support the execution of code out of RAM. In these systems the addressed memory is not a reliable source to identify an instruction fetch.

4.3.1.3 MPU interface

Beside the hardware interfaces the MPU needs a software interface to allow its interaction with software components. Since the peripheral units of MCUs are usually using memory mapped input/output (MMIO), a similar interface should be used. The functionality of the MPU interface depends on the implemented MPU memory variant. In case of using an external storage the interface must provide at least the registers listed in Table 4.3.

Table 4.3: Register interface of the MPU when using external storage.

Register	Size	Description
CTRL	8-bit	Common control register
SAID	8-bit	Write to set new SAID, read to get caller's SAID
ADDR	8-bit	Addresses the DDT entry
DATA	8-bit	Data register to read from and write data to the current DDT entry
DDTBR	n-bit	Base address of the DDT (size depends on IO address space)
SAIDBR	n-bit	Base address of the SAID stack

As the base address registers for the DDT and the SAID stack are security critical, write access must be restricted to trustworthy SAs. We use an MPU enable flag that can be set during the system's boot process. The flag activates access control operations or blocks all write operations to base address registers. In an alternative, more flexible configuration, the MPU registers can be assigned to a data space, which is accessible by a memory management SA only. An MPU-enable signal is still needed to setup data spaces during the boot process.

In case of using an internal memory, the DDT must be managed via the MMIO interface. This can be done by using the `ADDR` and the `DATA` register. Since each register that is directly accessible via MMIO consumes address space of the MCU, the size of the MMIO interface must be chosen with care. A minimal interface will provide only two registers: a command register and a data register. These two registers must provide the following operations:

- enable DDT entry,
- disable DDT entry, and
- update DDT entry.

Due to the fact that the size of a DDT entry is larger than 8 or 16 bit, the data must be read and written in a burst mode. First, the command register is written. Afterwards, the data register is read or written. The number of operations depends on the size of the data that are addressed by the operation. SAID management commands can be implemented by using the same register interface, so that two registers are sufficient.

4.3.2 Software-based activity isolation

A secure isolation can be enforced only by an additional layer that decouples the software from the underlying hardware. In the following, we discuss three approaches for an software-based implementation of a security nucleus for TSSs that are able to provide a secure isolation.

In contrast to a hardware-based memory protection, which is capable to control each memory access, a software-based approach is limited to controlling data access. As an all-embracing protection will cause an immense performance drawback, we introduced an extended control flow protection that observes destinations of each function call instead.

4.3.2.1 TSS-focused technology review

We identified three different techniques of a software-based isolation. First, SFI can be used to provide a sandboxing for SAs by monitoring memory access. Second, a secure isolation can be guaranteed by a virtualization of an ISA. Third, an approach of a partial emulation of software components can be used to overcome the drawbacks of the two other approaches.

Sandboxing

A sandboxing of SAs as introduced by Wahbe at al. [WLAG93] requires an identification of all critical operations. The SFI approach proposes a compiler extension or a static analysis of the binary code to detect critical instructions. Both approaches are applicable on TSS whereas a compiler extension gives more flexibility. In contrast to the safety related approach where write operations and function calls must be identified only in a secure system each memory operation must be controlled.

Listing 4.1 shows three examples of safe instructions. All register-register instructions are safe. Furthermore, an instruction is safe if the memory that is used for read and write is directly given. It can be an address or a label that can be resolved to an absolute address at compile-time.

Listing 4.1: Safe and unsafe assembler instructions on an MSP430.

```
# safe instructions
MOV.B R9, R12
MOV.W #0x000a, R14
CALL #__mpyl_f5hw

# unsafe instructions
MOV.B 0x000c(R13), R13
CALL @R15
POPM.A #1, R10
RETA
```

An instruction is unsafe, if it uses a register-indirect addressing mode. The addressing mode can be used for both data and code. Furthermore, each stack operation uses a register-indirect addressing mode with auto-increment. However, in case of using a compiler extension the number of unsafe operations can be reduced. Especially the stack operations can be replaced by global variables. The concept requires a considerable memory overhead but reduces the number of run-time checks in a significant manner.

Although the sandboxing technology can isolate an SA with a slight gain of the application execution time, the memory layout must be static. Any dynamic memory management operation can invalidate a static program analysis. An access to a previously valid memory section, as instruction two of Listing 4.1, may be invalid in case of a memory management operation,

which manipulates the accessed section. Such an operation will not be recognized by a static compile-time check.

Full system emulation

As sandboxing of critical operations requires a static memory layout, a data space management as proposed by our hardware implementation cannot be implemented. A highly flexible isolation can be provided by an instruction set emulation. But due to the lack of a hardware-based privilege separation in tiny scale MCUs common virtualization concepts cannot be used. Instead, a full system emulation is necessary. Otherwise, an adversary can occupy the CPU for an arbitrarily long time. But it is required that the security nucleus gets control over the CPU just before executing an unknown instruction.

However, in contrast to the SFI technology, a full system emulation allows an execution of binaries without any instrumentation. Furthermore, the emulated ISA has not to be identical to the one of the host system and the use of a light-weight and optimized ISA becomes feasible. The performance and the program size overhead of a full system emulation is its major shortcoming. In a rough estimation based on evaluation results of Bellard [Bel05], an emulated system will be about 100 times slower than native code.

Partial emulation

An alternative approach for a secure isolation of software modules on sensor nodes is given by Weerasinghe at al. [WC08]. Instead of a run-time checker as proposed by SFI or a full system emulation only critical memory operations are emulated. The approach proposes a virtual instruction set (VIS) based on the ISA of the host system. The VIS includes only instructions needed to replace critical memory operations. All other instructions are retained unchanged. For the enforcement of a secure isolation, the system must include an extended compilation model, a verifier, and an execution environment.

Compilation The software modules are compiled to the VIS, which is based on the host ISA but augmented by the emulated memory operations.

Verification An online verifier ensures that the module code does not include any native memory operations.

Execution The execution is dispatched to a small run-time environment when an emulated memory operation is encountered.

The VIS has to include replacements for each memory operation and each stack, call, and return instruction. To simplify the run-time environment the VIS is limited to load and store operations for any memory access and the stack, call and return instructions. The compiler must ensure that the remaining instructions use registers only. We will briefly introduce an example implementation of a partial emulation for an MSP430 in Section 5.1.3.

4.3.2.2 Guarded DDT

In contrast to a hardware-based DDT with a parallel look-up, a software-based SN has to search sequentially for a DDT entry. We propose the concept of guarded page tables, which allows an efficient entry look-up and supports pages with a variable page size. The guarded DDT requires a data space size with a power of two, which is given by the SIT strategy.

Although, a guarded DDT makes an efficient look-up possible the run-time overhead will be still substantially. Therefore, a DLB might be necessary. The run-time and code size overhead of a software-based DLB can be neglected. Due to the frequent use of the DDT look-up and the locality of TSS applications the achieved benefit by a DLB will be significant.

4.4 RBAC on tiny scale systems

A fine-grained access control can be defined by adapting the RBAC model for TSSs. We have introduced the model in Section 3.1.2 and have seen that the T-RBAC module was adapted to WSNs. A practical implementation of the T-RBAC module in enterprise information systems was presented by Sainan [Sai10]. But to the best of our knowledge an application of RBAC on TSSs was not presented yet. In the following section we present an adaptation of the RBAC model to TSSs.

Although TSSs process a tiny volume of data and their applications are clearly defined and mostly simple, a small number of individual SAs can always be identified. Figure 4.13 illustrates an example of an SA for receiving a network packet. Such an SA starts usually with an interrupt, raised by a GPIO line. The ISR invokes the network driver to handle the event. The driver copies the network packet from the transceiver IC to the data memory and invokes the protocol stack to process the packet. In a hypothetical example, the protocol stack could use a hash module to verify an HMAC included in the network packet. Finally, the actuator is invoked to execute the transmitted command.

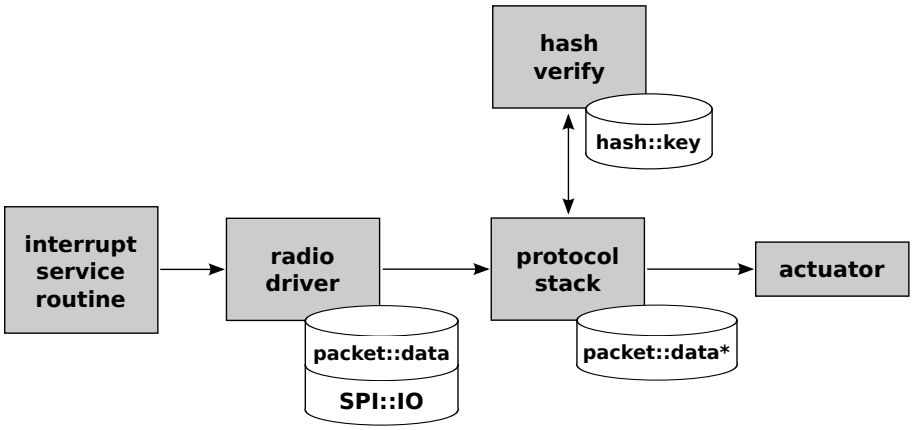

Fig. 4.13: An example of an event-driven software activity on a tiny scale embedded system.

The example illustrates steps that are necessary to process a network packet. In a well-defined software all these steps can be isolated and need access to the shared network object only. Furthermore, we can identify private data objects such as the MMIO resources for the communication to the transceiver IC and the hash key. All these data objects are

used by dedicated software modules and sharing them with other activities is not necessary. In addition, the program code can be assigned to specific software modules. Although we can use a DAC model to describe the example, in a more complex system the DAC model would provide a coarse-grained isolation only. If we extend the example by a software activity for sending a network packet, the protocol stack would have access to similar resources. We cannot differentiate between these two activities in the DAC model. Therefore, a more fine-grained isolation is necessary.

4.4.1 Application of RBAC terms to TSSs

We introduced our security enhanced platform for TSSs with the demand of a definition of SAs. Furthermore, we defined CDCs to transfer control among SAs. In the following, we will give an adaptation of the terms of the RBAC model to our terms already defined within the context of TSS applications.

User Software systems of TSSs usually do not identify any user. But we can identify SAs that perform a specific task. We have introduced the example of a network packet reception in this section. Similar activities can be defined for the two software examples introduced in Section 2.5 and further applications. In the following we treat an SA as a user.

Role The matching of SAs to users has led us to the adaptation of software components to the term role of the RBAC model. A software component is a software class or an object file of a modular program. Hence, in a TSS an SA takes a role by using a specific software component. Software components can be shared between different SAs. This is also common for roles in the RBAC model.

Session An application of sessions becomes necessary, if an SA has multiple execution paths. Each path may use an SA in different roles to perform given tasks. When using sessions, a very fine-grained, path-specific clearance becomes possible. We will map an execution path of an SA to a session[4].

Operation Software modules include attributes and methods, where a method is a function or a procedure. We match RBAC operations to software methods, so that we can assign a permission to an SA to invoke a method of a software module.

Object An object is matched on a data space as defined in Section 4.1. Since an SA has full access to a data space that it owns, we assign permissions to roles used by an SA to access data spaces. Furthermore, we can give access rights on data spaces by allowing an SA to perform an operation on behalf of a foreign SA that has access to the object.

[4]The labeling of execution paths within an SA must be done by additional software annotations.

When using the adaptation above, the RBAC model depicted in Figure 3.2 can be drawn for TSSs as shown in Figure 4.14. In contrast to common access control models, the RBAC narrows the access control down by assigning permissions to an SA to invoke a method or to access an object directly only when it performs a certain role.

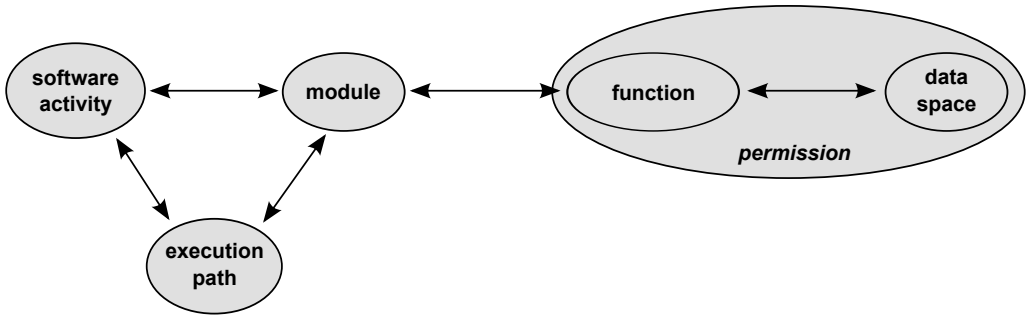

Fig. 4.14: An adaptation of the RBAC model to the terms of TSSs.

4.4.2 Security policy definition (SPD)

In the last sections we introduced basic principles: e.g. data spaces and SA flow integrity to implement a secure platform. Furthermore, we presented an adaptation of RBAC terms to the terms of TSSs. In this subsection we will introduce the security policy definition (SPD). The SPD includes the assignment of SAs, software modules, execution threads, operations and data spaces to the elements of a TSS. It defines access control and information flow between these elements. Although an enforcement of a security policy may also be possible on firmware binaries, the achievable granularity may be small and the technical overhead would be very high. Therefore, we are convinced that a fine-grained SPD should be realized on source code.

Schneider points that a single large monolithic SPD is difficult to comprehend and to maintain, so that real system security policies are best given as collections of simpler policies [Sch00]. Therefore, we propose a system's SPD that is the result of composing the policies with different application fields. Following this approach, we propose an SPD that consists of two parts: the *security policy book (SPB)* and *source code annotations*.

Composing the SPB and the source code annotations is done by our compilation model as described in Section 4.4.3. In the following we give a brief introduction to the SPB and the source code annotations according to the RBAC terms.

4.4.2.1 Security policy book (SPB)

We introduced the SPB as the application-specific component of the SPD. The SPB should be defined in a separate file independent from the program sources. We motivate this decision by the fact that the program sources may be used for multiple projects with different security requirements. As our approach does not allow any run-time modifications of the SPD and the overhead of the enforcement of security policies depends significantly on its granularity SPDs must be integrated into the TSS application build process. Using a separate, application-specific SPB helps to cope with all these requirements.

The SPB forms the application-specific part of the SPD. Similar to the work of Loscocco et al. [LS01] the SPB contains four kinds of statements: user and session declarations, role declarations, role assignments, and user transitions.

Users and sessions are labeled with a symbolic name. The symbolic name is mapped to a software module or a function of a TSS application that initiates a new context.

```
user USER <- filename<::function>
```

Roles are also labeled with a symbolic name and mapped to software modules that implement the role. Since a role may be assigned to a number of software modules, it is mapped to a list of modules.

```
role ROLE <- {filename1, filename2, ...}
```

Role assignments are mappings of the symbolic names of users and sessions to the symbolic names of roles. A user can have multiple roles, so that the mapping includes a list of roles.

```
assign USER <- {ROLE1, ROLE2, ...}
```

User transitions specify allowed transitions between users by defining a unidirectional relation between a symbolic user name and a list of symbolic user names.[5]

```
transition USER1 <- {USER2, USER3, ...}
```

The SPB definitions cover the RBAC terms user, roles, and their relations. Operations and data objects are described by software code annotations and are omitted within the SPB. We are convinced that the terms defined by the SPB give a software engineer full control on the granularity of the SPD and the ACL can be automatically applied on users, objects, and roles by the extended compilation model.

4.4.2.2 Source code annotation

The standard of programming languages does not support a binding of security policies. But source code annotations can be used to couple the security rule set with the functional set of the TSS software. Since source code annotations will be part of all applications, it must be possible to mask them, if they are not used. We can identify two basic technologies for source code annotations suitable for our purpose.

[5] Transitions are not part of the basic RBAC model. They were introduced by Youman to provide a higher level of security [You96]. Transitions are implemented by grsecurity in the Linux OS [Spe05].

Without syntax support Annotations are implemented by using naming conventions. The documentation tool `doxygen` uses code annotations that are integrated in source code comments. The sources are processed by an additional tool that compiles the annotations in a final document.

With syntax support Annotations include additional information for the code generation, optimization and linking process without any extensions of the build tools. E.g. the GCC compiler supports an additional keyword `__attributes__`, which can be used for custom purpose.

The SPB holds information about software activities, modules, and execution threads. To complete the RBAC model, operations must be described by source code annotations. We have mapped operations to functions, which can be labeled by source code annotations. We assume that all functions of a module are private if no other information is available. Thus a software developer has to annotate only the public functions of a module.

We use source code annotations without syntax support. Annotations without syntax support offer more latitudes, so that we are able to cope with our requirements. Source code annotations require an additional processing step within the compilation model. A similar approach is used by safe languages, as introduced in Section 3.3.3. Since public function calls are CDCs, a detailed description of the function signatures is also required. Therefore, source code annotations must provide additional information regarding function parameters and the return value. Especially, *call by reference* requires additional information about the referred object for applying marshalling if needed. Furthermore, input, output, and inout parameters must be identified to make marshalling optimizations possible.

A public C function may be annotated by the grammar defined in Listing 4.2. We extended the parameter definition so that we get additional information about the direction and in case of references the size of the referred object. The defined grammer extends the C function signature without changing the basic syntax.

Listing 4.2: Annotation grammar for public C functions.

```
<function>  := PUBLIC <funcdef>

<funcdef>   := <type> <funcname>(<paramlist>) | <type> <funcname>()
<paramlist> := <param> | <param>, <paramlist>
<param>     := PARAM(<type> <varname>, <direc>)
             | PARAM(<type> <varname>, <direc>, <size>)
<direc>     := IN | OUT | INOUT
<size>      := <number> | <macro>

<type>      := /* any data type of C */
<funcname>  := /* function name */
<varname>   := /* variable name */
<number>    := /* any unsigned number */
<macro>     := /* C preprocessor macro that returns a number */
```

We do not restrict the type of parameters. Object parameters with references require special care. These parameters require a deep copy operation, which is very difficult to handle. Therefore, the current implemented scheme does not allow parameters that require a deep

copy operation. The programmer has to adapt the structure of an object manually in such a way that it does not contain any references.

4.4.3 Compilation model

For a proper integration of the SPD we propose a three-step compilation model as illustrated in Figure 4.15. In the first step the SPB and the annotated source code are processed by the security policy compiler (SPC).

The SPC is the core component of the extended compilation model. It generates native sources, which can be compiled with a native compiler. The generated sources include all CDCs, region grant, and region map operations. The CDCs include the ACL that is derived from the rules defined in the SPB and the program sources. We propose an automatic ACL generation based on a CFG analysis as described by Abadi et al. [ABEL05a].

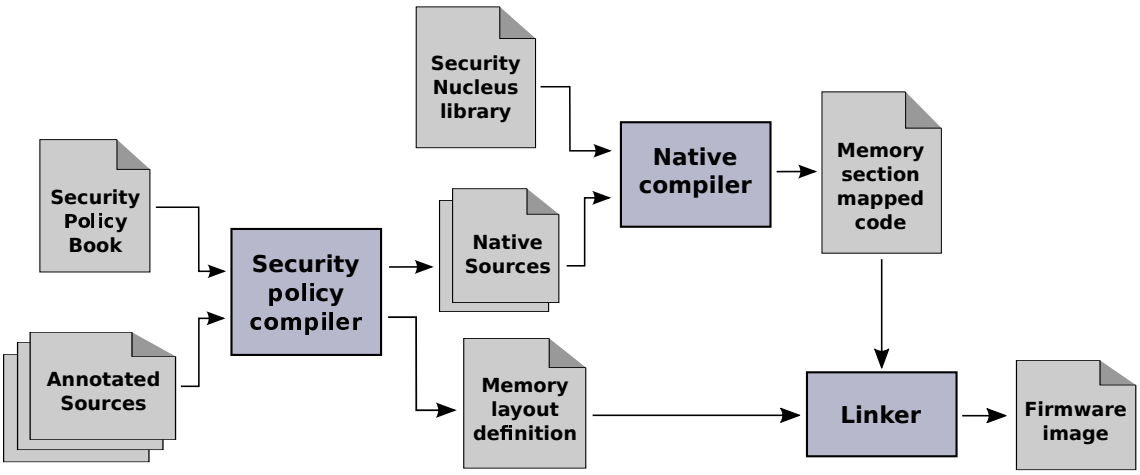

Fig. 4.15: Three-step compilation model of the TSS security platform.

Beside the CDCs and the ACL, the SPC must generate a memory layout definition. The memory layout is stored in a separate file and is needed by the linker to relocate the position-independent output of the native compiler within the final linking step. The layout file includes definitions of memory sections that represent the protection domains defined by the SPB. The native sources include the mappings of the software components to their protection domains. The final binding is done by the linker, which copies the components to the defined memory sections in the final firmware file.

The extended compilation model is not OS-specific. It can be applied to each OS library, as introduced in Section 3.4.4.2. But the SPC is strongly coupled with the SN and processes the SPD of the TSS application. Although the SN might be implemented in a separated source code library that can be compiled by the native compiler, the SPC, the SPD, and SN form a functional package. The CDCs including the ACL, the grant and the map operation are generated by the SPC based on the SPD and their implementations are an essential part of the SN.

We will give detailed descriptions of two examples of our compilation model in Section 6.2. We implemented the model successfully for the langOS OS. We integrate source code annotations and a SPC into the OS build chain and applied it to the Meetering app. In addition,

we will present an extension of the configurable compiler suite CoMet that implements our compilation model for the tinyVLIW8 soft-core processor.

CHAPTER 5

Assembling the security nucleus

A secure isolation of software activities is mainly influenced by the assembling of the security nucleus. This chapter gives an introduction to the assembling of the hardware-based and the software-based concept of the SN. We start our description with a more detailed view on two real systems used by our example applications introduced in Section 2.5.

We introduced OSs for embedded systems in Section 3.4.4.2. All these systems provide a subset of basic primitives of common system kernels. They use compile-time optimizations, which take the special properties of TSSs into account. We propose an assembling of our security nucleus that follows the same design philosophy as embedded systems do. We divided the SN in a hardware-based memory protection nucleus[1] and a compile-time generated nucleus gate, as illustrated in Figure 5.1.

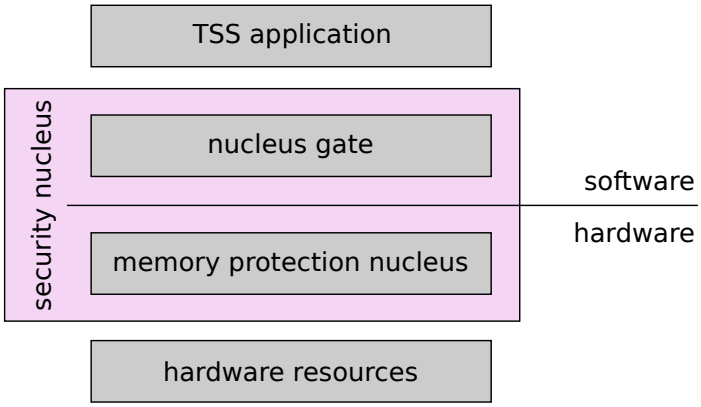

Fig. 5.1: The security nucleus divided in a memory protection nucleus and the nucleus gate.

We implemented the nucleus gate as a software component that features CDCs including access control and data space management. Application-specific components of the nucleus gate are generated at compile-time. Hence, its implementation is resource and performance optimized. Furthermore, the nucleus gate includes the interface to the memory protection nucleus. The memory protection nucleus enforces resource isolation and can be implemented in hardware or in software.

This chapter illustrates the assembling of the memory protection nucleus. We describe the implementation of a hardware-based, tailor-made MPU for both processor architectures. Both implementations are very similar, so that we give a combined description but expose differences. Afterwards, we briefly introduce the implementation of the software-based approach on an MSP430. Finally, we will describe the software components of the nucleus gate.

[1] In case of a software-based isolation a tiny hypervisor will be used to emulate the hardware-based memory protection nucleus.

5.1 The memory protection nucleus

The memory protection nucleus has to enforce SA isolation, so that an SA can access only those resources that are assigned to it. The implementation of the memory protection nucleus is tightly coupled with the processor's ISA. Therefore, we will introduce two processor architectures of TSSs first. We use these architectures to evaluate our implementation later.

This section will give a brief introduction of these two processor architectures. We focus on instruction fetch, data memory access, and interrupt handling. These operations are crucial for our MPU integration. More detailed descriptions of the processor architectures are given by Texas Instruments for the MSP430 [Ins06] and by Stecklina et al. for the tinyVLIW8 in [SM14].

5.1.1 Processor architectures

We have implemented our SN on an IHP430X and a tinyVLIW8 soft-core processor. The IHP430X implements the von-Neumann architecture and the tinyVLIW8 is based on the Harvard architecture. Both architectures are common in MCUs and have already been introduced in Section 2.4.1. By covering both processor architectures, we show that there are no principle obstacles for applying your concept to any other MCU.

5.1.1.1 Von-Neumann architecture - IHP430X

The MSP430 is a mixed-signal MCU introduced by Texas Instruments (TI) and widely used in medical, industrial, and consumer devices. The IHP430X is an MCU built around a soft-core processor developed by Fraunhofer IPMS [Grä10]. The ISA of the IHP430X is binary compatible with the 20-bit MSP430X architecture. It is equipped with serial interfaces, an A/D converter, Timer, GPIO, and a crypto unit that includes a SHA1, an AES, and an ECC core. The compatibility enables the use of the MSP430 compiler suites, e.g. the MSP430 GNU compiler collection (GCC) or the Code Composer Studio (CCS) provided by TI. Beside the IHP430X MCU the openMSP and the NEO430 soft-core processors are available at openCores [Gir10, Nol15]. Both cores implement the MSP430 16-bit ISA and are public available under the BSD license.

The MSP430 processor core has a von-Neumann architecture with a shared code and data memory bus. Furthermore, as illustrated in Figure 5.2, all peripheral units are connected to the same memory bus. Therefore, the MSP430 has a single address space that includes SFRs, MMIO, RAM, and ROM. The program code should be located in the ROM section but can be executed from the RAM as well[2]. Therefore, the MCU is vulnerable for local attacks, which makes it to an easy target of malicious users [Goo07, Goo08].

The MSP430 has a 16-bit MAB and all internal registers have a width of 16 bits. Due to the shared architecture the small address space is a challenging limitation. Therefore, the MSP430X architecture with a 20-bit address bus has been introduced. It is binary compatible to the 16-bit architecture, which means that 16-bit programs can be used unchanged. But internal registers and the MAB were extended to 20 bits. The extended address space can

[2] Code execution from RAM is not supported by the openMSP soft-core processor. The core demands a location of the program code within the flash memory section.

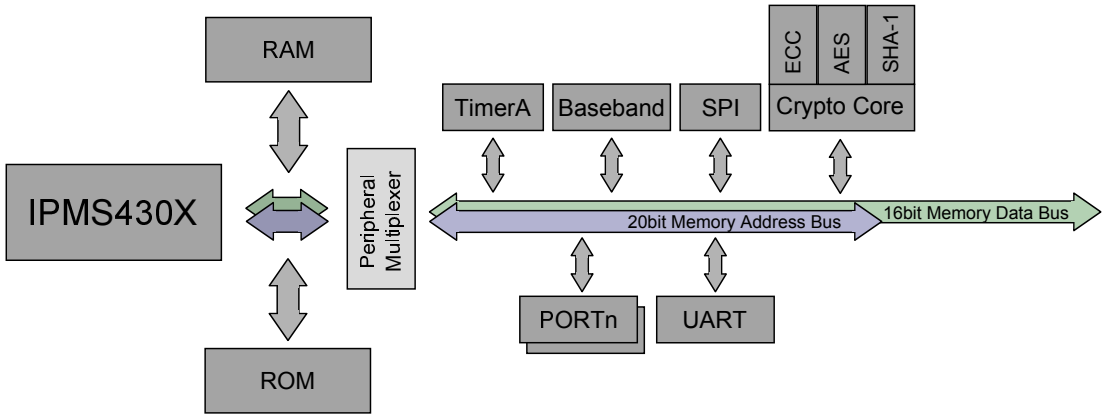

Fig. 5.2: Block diagram of the IHP430X MCU. As the von-Neumann architecture connects memories and peripheral units with the processor core by a sole memory bus, all resources are mapped in a single address space.

be used by additional instructions, which require a compiler suite that supports the MSP430X architecture.

Instruction decoder

The MSP430 is marked as a RISC system because it has 27 different instructions only. But in contrast to ”real” RISC systems the MSP430 supports seven addressing modes, which can be combined with each instruction. Furthermore, all instructions are coded in 16-bit words and the number of words per instruction differs from of a single word up to four words. The instruction length depends on the instruction format[3] and addressing mode. Therefore, the instruction fetch of the MSP430 is much more complex than expected for a RISC system.

The processor core fetches an instruction in single 16-bit words. After each fetch the instruction snippet is decoded. Afterwards, either an additional load is performed in case of a multi-word instruction or the instruction execution is started. The instruction fetch of a 16-bit word and its decoding is done within a single clock cycle. The following execution steps may require a multiple clock cycles and depend on the instruction. The final execution time of a single instruction differs from a single cycle up to seven clock cycles.

Addressing modes

The MSP430 supports seven different address modes for the source and four addressing modes for the destination operand [Ins06]. Table 5.1 lists the different addressing modes of the MSP430. The register mode and the immediate mode do not include any memory access. An implementation of a software-based MPU has to consider five source and four destination addressing modes.

We will introduce our tiny hypervisor in Section 5.1.3. The VIS and the binary rewriter are tightly coupled with the ISA of the MCU. Especially, the available addressing modes are a major factor for the complexity of the tiny hypervisor implementation. We use the MSP430 to illustrate our approach and considering its seven addressing modes.

[3]The MSP430 has three core instruction formats: single-operand, double-operand, and jump instructions.

Table 5.1: Source (As) and destination (Ad) operand addressing modes of an MSP430.

Addressing Mode	As	Ad	Description
Register mode	x	x	Register contents are operands
Indexed mode	x	x	Register plus offset points to the operand
Symbolic mode	x	x	PC plus offset points to the operand
Absolute mode	x	x	The instruction includes an absolute address
Indirect register mode	x	-	Register is used as a pointer to the operand
Indirect autoincrement	x	-	Similar to the indirect register mode but the register is incremented afterwards
Immediate mode	x	-	The instruction includes an immediate constant

Interrupt handling

The MSP430 uses interrupts, where each ISR has its own vector stored in the interrupt vector table (IVT). The IVT has a fixed size and is located at the end of the 16-bit address space. The ISR can be located anywhere in the 16-bit address space. Interrupts are enabled by setting the global interrupt enable (GIE) bit in the processor status register and within the peripheral unit that drives the interrupt.

Interrupt processing starts by completing the current instruction or by enabling the master clock when the CPU is off. In a first action of interrupt processing, the processor status register and the current program counter are pushed onto the stack. After saving the registers the highest priority interrupt is selected. Finally the processor fetches the interrupt vector and stores it in the program counter so that the user-defined ISR is executed within the next instruction. When the interrupt processing, is finished the program returns from the interrupt by executing the `reti` instruction. In contrast to an ordinary return instruction `reti` pops the status register and the program counter from the stack.

Interrupt handling is globally controlled by the GIE bit in the status register. Only if it is set a pending interrupt is recognized and its ISR is called. In case of multiple interrupts the one highest priority is handled first. On entering the ISR the GIE bit is cleared, so that interrupt requests of all priorities remain pending until the GIE bit is set again. On modern MSP430 families the user can manually enable the GIE bit within an ISR, so that nested interrupts are possible. But most users do not allow nested interrupt because its handling is very complex. A sequential interrupt handling is in common use instead. At the latest on interrupt return the GIE bit is restored.

Implications for an MPU integration

Due to the complex instruction fetch and the different execution time of instructions an integration of a tailor-made MPU is not possible without knowledge about the internal execution stage. Therefore, an additional fetch signal from the instruction decoder of the processor core to the MPU is necessary.

The MSP430 does not support a separate interrupt stack and uses the current program stack during interrupt handling. Furthermore, the `reti` instruction is executed within the interrupt and needs access to the saved status register and the program counter. Hence, the interrupt handling must be implemented with special care to permit access to the saved data without harming the isolation scheme.

5.1.1.2 Harvard architecture - tinyVLIW8

The tinyVLIW8 is an 8-bit embedded controller with a Harvard architecture. The controller uses hardware-separated address spaces for the instruction memory, the data memory and the I/O resources. As shown in Figure 5.3, each address space is accessed by an individual bus. The processor core is a RISC system with a load/store architecture.

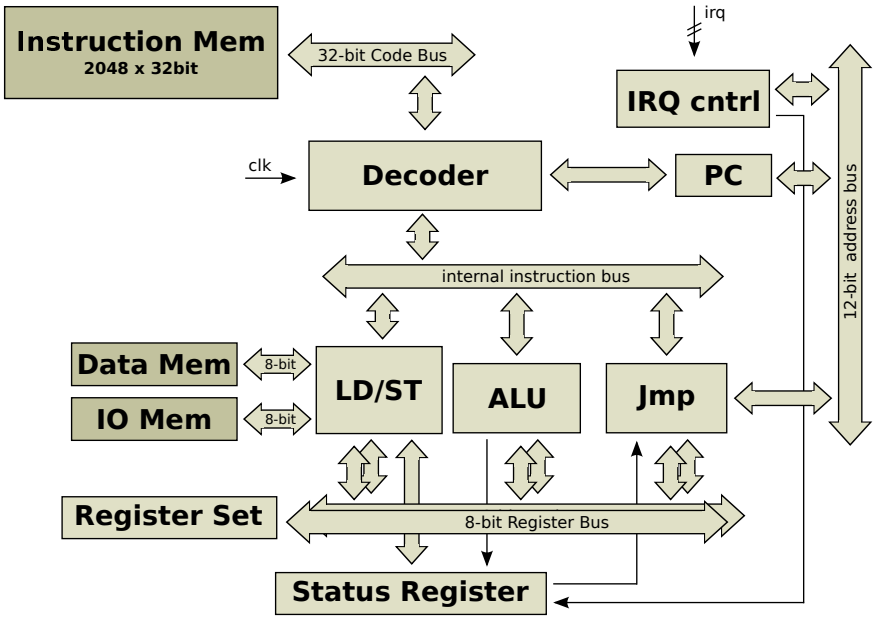

Fig. 5.3: Block diagram of the tinyVLIW8 processor core.

The data memory and the I/O memory are accessed by load and store instructions only. All values needed for an operation must be loaded from data or I/O memory to be present in a register. The result of an operation is always stored in a register. It must be written back to the memory by an additional operation. Due to the load/store architecture, each instruction can be coded in a single 16-bit word and is always executed within two clock cycles. Hence, the tinyVLIW8 has a very small design footprint and a predictable behavior, which predestines it for embedded control tasks in TSSs [SM14].

VLIW instructions

As a very large instruction word (VLIW) processor the tinyVLIW8 executes two instructions in parallel if they address different functional units. As illustrated in Figure 5.3, the processor features three functional units: *LD/ST*, implements all load/store operations from and to the data and the I/O memory, *ALU*, implements all register-register operations as arithmetic, logical, shift, and move operations, and *JMP*, implements conditional and unconditional jumps.

Two 16-bit instructions are coded in a single 32-bit instruction word. Hence, the instruction memory data bus (MDB) has a width of 32-bit, so that each instruction memory address addresses a 32-bit VLIW instruction. Since the instruction memory always returns a 32-bit instruction word a single 16-bit instruction cannot be addressed. To implement a single 16-bit instruction, the same instruction opcode can be copied into the second 16-bit instruction of the same 32-bit instruction word and the processor will ignore it.

Instruction execution

The instruction execution of the tinyVLIW8 soft-core processor is divided in four phases: *fetch*, *decode*, *execute* and *write back*. Each instruction phase is executed within half a clock cycle. The processor core does not feature an instruction pipeline. Therefore, the instruction memory is accessed during the fetch phase only. The data and the I/O memory are read during the execute phase or written during the write back phase. An instruction memory read access is shown in Figure 5.4.

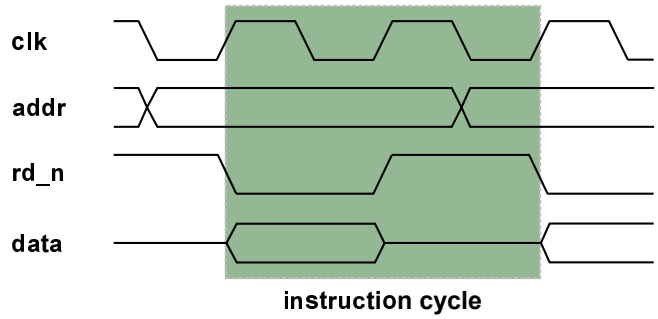

Fig. 5.4: Instruction memory access of the tinyVLIW8 processor core.

The processor provides the address available at the beginning of the last half of clock cycle of the previous instruction. The instruction word is fetched with the next rising clock edge. The data must be stable at the next falling clock edge. An access on the data memory and the IO peripherals works similar. The address is always available half a clock cycle before the access starts. To avoid a processor stall it must be guaranteed that the data are available in the same manner when integrating an MPU.

The instruction memory uses an 11-bit memory address bus (MAB) and the data and the IO memory use an 8-bit MAB. The instruction memory is accessed by the processor core during the instruction fetch phase only. The data and the IO memory are used during the decode and the write back phase.

Interrupt handling

The tinyVLIW8 soft-core processor is optimized for low latency interrupt handling. It provides a fast interrupt handling. Similar to the MSP430, the soft-core processor uses vectored interrupts stored in an IVT. The IVT is located at the end of the 11-bit address space. But the interrupt vector of the tinyVLIW8 contains a 32-bit instruction word instead of an address of the ISR. An ISR is invoked by placing an instruction word that contains a jump instruction in the IVT. The second 16-bit instruction beside the jump instruction can contain an instruction that is already part of the ISR. The instruction placed in the IVT is executed within the interrupt context.

To reduce the interrupt overhead the tinyVLIW8 processor is equipped with a shadow register bank. All general-purpose registers except two are switched during interrupt handling. Therefore, the ISR has not to save the registers. The remaining two registers are used for the stack pointer and the stack frame, so that the ISR can share the stack with the user application. The program counter is automatically saved within the jump unit and has not to be saved as well. A *return-from-interrupt* instruction is implemented by copying the saved program counter into the processor's program counter. A separate *return-from-interrupt* instruction is not available.

The processor's status register has an *in-interrupt* flag. It is set when an interrupt occurs and will be automatically cleared when the program counter is loaded.

5.1.2 Tailor-made hardware-based MPU

An initial approach of our tailor-made MPU has been implemented in SystemC by H. Menzel in his master's thesis [Men10]. The design has been integrated in the hybrid-simulation environment for the MSP430 (HSE430), see Section 7.2.2.1 [SLM11]. In a following master's thesis supervised by me, the design has been evaluated by E. Bergmann by implementing a software framework for an isolation of SAs [Ber12]. Based on these two master's theses and the evaluated requirements of TSSs applications we finalized our design. We consolidated our results by implementing an MPU for the tinyVLIW8 soft-core processor. In addition, we adapted the implementation for the MSP430. In the following we will describe an implementation of a tailor-made MPU and its integration in the soft-core processors.

5.1.2.1 Definition of a DDT entry

Section 4.1.1 lists four different strategies for the data space boundaries description. We mentioned that the chosen strategy depends on the location of the DDT and the complexity of the DDT entry look-up. Since all compare operations need a large amount of logic in hardware the size in 2^n (SIT) strategy, proposed in Section 4.1.1.2, will result in the smallest hardware design. Therefore, we have chosen the SIT strategy to describe the segment boundaries in our hardware-based MPU. The number of bits needed to describe the boundaries of a segment on the IHP430X and on the tinyVLIW8 is shown in Table 5.2.

Table 5.2: Size of the segment boundaries descriptor in the DDT entry on an IHP430X and on a tinyVLIW8 when using the SIT strategy.

	base address	segment size order	maximal segment size	$\sum$
IHP430X	20 bits	5 bits	2^{31} bytes	25 bits
tinyVLIW8	11 bits	4 bits	2^{15} bytes	15 bits

The base address size is given by the size of the address spaces of the processor architecture. The IHP430X features a 20-bit address space and the tinyVLIW8 features an 11-bit one, which results in a 20-bit base address for the IHP430x and an 11-bit base address for the tinyVLIW8. The segment size is coded by a power of two order and the order is stored in the segment size order field. The number of bits used for the size field is much smaller than the base address field size. In case of using an external memory that is accessed by the MDB, the MDB width defines a suitable size of a DDT entry. The IHP430X has a 16-bit MDB[4] and the tinyVLIW8 has a 8-bit MDB, so that the size of a DDT entry has to be applied to the MDB width. Hence, at least 32 bits are needed for the IHP430X and 24 bits are needed for the tinyVLIW8.

[4]The IHP430X can access 8-bit data, but implements internally a 16-bit MDB. Therefore, on each data access a 16-bit data word is loaded. An 8-bit access is not implemented.

Software activity identifier

Following the description in Section 4.1.1 each DDT entry must contain the segment boundaries, an owner, and the capabilities. The owner of a memory region is identified by its SAID. Investigations have shown that the number of SAs in a TSS application is quite small [Men10]. Hence, we are convinced that a 4-bit SAID is suitable for a broad variety of TSS applications.

The 4-bit owner field allows 16 different SAs in a TSS application. Due to the interrupt implementation of both processors, one identifier with a special purpose has to be defined. When an interrupt occurs both processors access the IVT to load the ISR address. But the interrupt event is asynchronous, so that it will be done at any time within the context of any SA. Thus, all SAs need access to the IVT and to a minimal code section to handling the interrupt event. Hence, we introduced a shared segment identified by the SAID *zero*.

A publicly available segment is implemented by using the zero SAID. A data space can be accessed if the current SAID matches with the owner field, with the SAID stored in the capability field, or one of these fields is zero. The permissions stored in the capability field are checked in case the owner field does not match or the owner field value is different from zero. It is common to implement a public data space by granting the data space to SA zero. The public data space can be protected from malicious access by applying appropriate rights. The data space owner, who has to be different from zero, can modify the data space at any time.

Capability field

The capability field of a DDT entry includes the SAID and the permission rights. We defined seven different permission rights. Therefore, the capability field needs at least 11 bits, seven permission bits and four SAID bits. An enabled permission mask bit gives the SA identified by the SAID the permission to use the the resources within the data space boundaries. We do not differentiate between data, code, or IO resources, the type of the addressed resource is invisible for the MPU.

In case of using an external memory, which is accessed by the MDB, an additional bit between the SAID and the permission rights is necessary to pad the size of the owner field and the capability field to a multiple of 8 bits. When using an internal memory with a flexible word size the fields can be stored without spaces. The register layout can be arranged for an application on read access by shifting the content accordingly.

5.1.2.2 DDT memory configuration

In both architectures, in the IHP430X and in the tinyVLIW8, the proposed MPU can be implemented as peripheral unit with an MMIO interface. Due to their different memory configurations we chose different configurations for the DDT memory as well. The IHP430X has a 20-bit address space and is usually equipped with kilobytes of physical memory. The tinyVLIW8 has an 11-bit code address space, an 8-bit data address space[5] and an 8-bit IO address space. By mapping these three address spaces in a single 12-bit address space a sole DDT became feasible.

[5]The memory of the tinyVLIW8 can be extended by using segments addressed by an additional segment register. In this configuration the data memory is limited to 32 kB + 128 bytes. But memory segments are not yet implemented. Therefore, we used the basic memory layout during this work.

Table 5.3 shows the size of a DDT entry for both architectures. Depending on the chosen memory configuration, the raw entry must be padded to fix the processor's word size in the memory. The raw size of a DDT entry includes the data space boundaries, the owner, the capability field, and the additional status flag, which indicates that a data space was mapped.

Table 5.3: Size of a DDT entry on an IHP430X and on a tinyVLIW8 with raw size and padded size in RAM.

	base address size	word size	DDT entry size (raw)	DDT entry size (in RAM)
IHP430X	20 bits	16 bits	42 bits	48 bits
tinyVLIW8	12 bits	8 bits	32 bits	32 bits

tinyVLIW8 with internal memory

The tinyVLIW8 processor features three different memory buses. To use a sole DDT the address space was extended to 12 bits. The most significant bit (MSB) of the address field in the DDT entry determines the memory, '0' instruction memory and '1' data and IO memory. The data memory and the IO memory are identified by the eleventh bit. Hence, the data memory has the prefix '10'[6] and IO resources the prefix '11'.

The DDT can be implemented in an FPGA by registers[7] or by memory blocks. In case of implementing a CAM-based look-up engine only registers can be used. The number of registers can be estimated by:

$$regNum = elemBits \times elemNum.$$

The usage of a CAM-based look-up engine is very expensive. E.g. in case of using 32 entries, the DDT consumes 1024 registers. In comparison to the size of the tinyVLIW8 processor, which uses only 500 registers, the number of registers is tripled. We give detailed measurement results of a real implementation in Section 7.2.1.

IHP430X with external memory

The IHP430X features a 20-bit address space. Due to the larger address space the size of a DDT entry is at least 8 bits larger than that of the tinyVLIW8 soft-core processor. In addition, the number of DDT entries has to be larger than 32 to support typical IHP430X applications. Therefore, we chose to place the DDT in a memory instead of in registers. Furthermore, we propose the use of the system memory of the IHP430X to store the DDT.

The shared use of memory makes the memory controller more complex. On the other hand, the DDT initialization can be simplified significantly. The memory can be accessed directly whenever the MPU is not enabled. Otherwise, the memory is protected by the MPU and can be accessed by it exclusively. We give an example of the DDT management on an IHP430X in Section 6.1.3.

[6] The DDT entry provides 10 bits for the data memory. Currently, only eight of them are used, so that a future memory extension can be mapped to the same DDT entry structure.

[7] Each logic element of an Altera Cyclone FPGA includes a programmable register that is implemented by a flip-flops. Hence, the number of registers is equal to the number of bits stored within logic elements of an FPGA [Alt12].

5.1.2.3 MPU placing

Although our MPU is designed as a peripheral unit, it has to be placed logically between the processor core and the resources. It is highly important to ensure that any access is passed via the MPU in a way that the data transfer can be blocked in case of an access violation. But an integration of the MPU in the MDB is not straightforward and will affect the timing of a memory access in a significant manner. To enforce our logical structure we exploited the fact that the IHP430x and the tinyVLIW8 soft-core processors use enable signals to address their resources. As shown in Figure 5.5 we added an additional combinatorial element to the peripheral enable signals `RdEn_n` and `WrEn_n`.

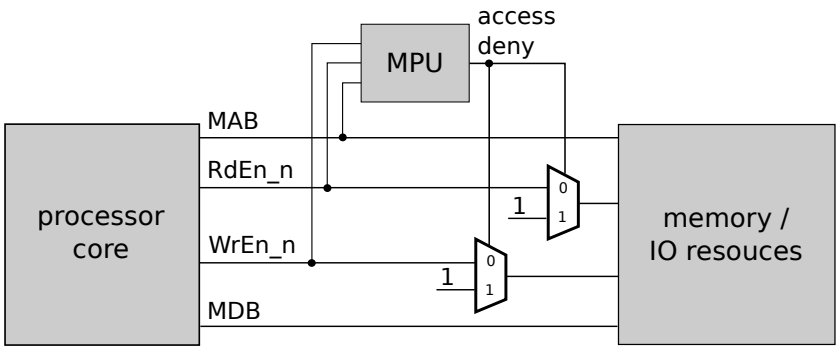

Fig. 5.5: The MPU controls access to the memory resources by overwriting the read and write enable signals.

All peripheral units as well as the memories of the soft-core processors are controlled by enable signals, which are pulled down if a unit is accessed. By overwriting the signals, a data transfer can be physically blocked. In case of an access violation the MPU keeps the signal on high level, so that the requested transfer is not recognized by the addressed unit.

In case of using an external DDT memory the MPU has to monitor any access into the DDT area. When the MPU is enabled any write access has to be blocked. A manipulation of the DDT may be done only via the MPU's MMIO interface.

5.1.2.4 Violation handling

We differentiate between memory access violations and data management violations. A memory access violation is raised in case of an illegal memory access, as read, write, or execute. A data management violation is raised in case of executing an illegal DDT management operation, as map, grant, append, or shrink. Both violations must be handled by the processor, but are signaled in differents ways.

Memory access violation

Due to placing the MPU between the processor core and the resources any read, write, and execute operation can be controlled. In case of an access violation an interrupt is raised and the processor stops the current execution and jumps immediately into the ISR. The processor's waveform of an instruction execution on a tinyVLIW8 is given in Figure 5.6. It shows the execution of a legal IO write instruction at address 0x01e and an illegal write operation at address 0x01f. The illegal instruction raises an interrupt `mpuIrq_s` that is handled by the

next instruction at address 0x7fc. The Figure also shows that the IO lines controlled by signals `mpuIoEn_n_s` and `mpuIoWr_n_s` stay high during the illegal access. The screen shot of the signal waveform of the illegal instruction is shown in Figure C.1 in Appendix C.2.1. The object dump and the assembler code of the application is listed in Appendix C.2.2 in Listing C.2.

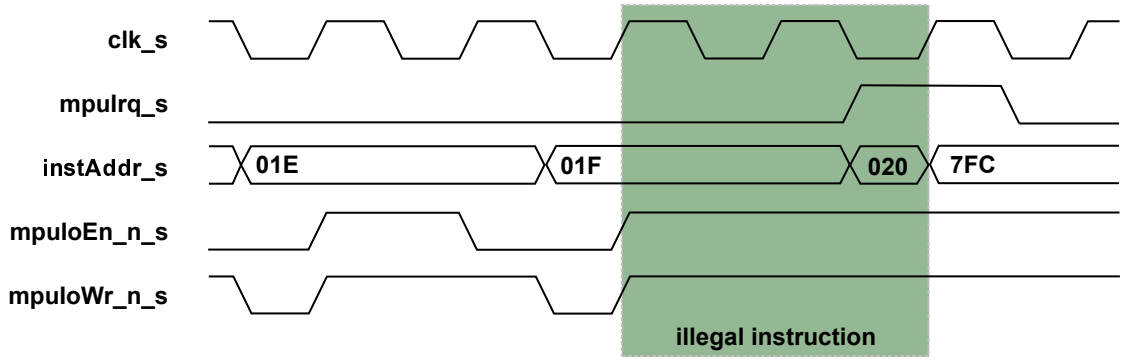

Fig. 5.6: tinyVLIW8 signal waveform of three instructions, where the second instruction causes a memory access violation. In the third instruction the processor core loads the ISR.

Since the instruction causing the violation cannot be passed directly to the interrupt handling an interrupt flag register is offered by the MPU. In case of an access violation the corresponding flag `rwx` is set within the register, so that within the ISR the register can be read to identify the causing instruction. The address of the causing instruction is stored automatically by the processor in the interrupt unit and can be read within the ISR as well.

DDT management violation

In addition, we defined the DDT management operations map, grant, append, and shrink, which are executed via the MMIO interface of the MPU. In case of an invalid operation, e.g. an SA tries to map a data space without permissions, the operation is aborted and a violation is signaled.

We can assume that a DDT management operation is a controlled and purposeful action. Hence, an interrupt is not necessary to signal an access violation. Rather the bad operation flag is set in the MPU register. The program has to check the register to ensure that an operation was successful. We chose that implementation because it simplifies the hardware of the MPU significantly and requires a minimal software overhead.

5.1.2.5 MMIO interface

We mentioned an MMIO interface for the MPU in Section 4.3.1. Due to the limited address space of the soft-core processors, the DDT cannot be fully mapped. In case of using the processor's memory the DDT initialization can be done directly. But when the MPU is enabled the DDT can be manipulated only via the MMIO interface and in case of using an MPU internal memory any DDT operation must be performed via the MMIO interface. Hence, the MMIO interface is required in both variants and its design is constrained by the limitations of TSSs.

MPU register interface

The tinyVLIW8 soft-core processor uses an internal DDT memory. The MMIO interface is defined by the registers listed in Table 5.4. Base registers are not needed, any DDT operation has to be done via the `ADDR` and the `DATA` register.

Table 5.4: MPU register interface of the tinyVLIW8 soft-core processor.

Register	Description
CTRL	Common control register
SAID	Currently active software activity
ADDR	Address register of the DDT internal memory
DATA	Data written to the address specified by the ADDR register

CTRL The `CTRL` register includes the MPU enable flag `en` and the status flags `op` and `ifg`. The register layout is shown in Figure 5.7. As mentioned before the status flags indicate an access violation. The `op` bit is set in case of a DDT management violation and the interrupt flags `ifg` are set in case of a memory access violation. The flags are cleared on a register read access.

SAID The `SAID` register implements the domain switch. On each write access the given SAID is stored on the SAID stack and becomes the current SAID. As introduced in Section 5.1.2.1 the SAID zero is used with special purpose. We use it to implement a *CDC return*. A CDC return is performed when writing the SAID zero into the `SAID` register. On read access the `SAID` register returns the SAID of the previous SA. The previous SAID is needed during each access control check, which is part of the software-based nucleus gate. The current SAID cannot be read by software. It is accessed by the hardware-based MPU only.

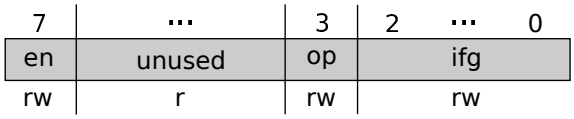

Fig. 5.7: The MPU control register is a 8-bit register that includes an enable flag and the interrupt flags.

DDT management operations

The DDT can be accessed by the `ADDR` and the `DATA` registers. We use the MSB of the `ADDR` register to differentiate between read and write operations. The lower seven bits of the address register specify the DDT entry. Therefore, the size of the DDT is limited to 128 entries. The number of DDT entries is a trade-off between memory and flexibility. In the context of TSS applications we are convinced that 128 entries is a suitable number.

Since a DDT entry includes more than eight bits, data must be transferred in a burst mode. First, the index of the addressed entry must be written into the `ADDR` registers. Afterwards, the burst is started by performing a read or write operation on the `DATA` register. The burst stops when all data are transferred or just after writing into the `ADDR` register. To avoid an

accidentally overwrite of another entry a burst of multiple entries is not allowed when the MPU is enabled.

The number of bytes and the structure of a burst depends on the operation that is performed. Each operation starts with a header byte and is followed by an operation-specific number of data bytes. An overview of the defined operations and their number of bytes are given in Table 5.5.

Table 5.5: DDT management operations.

Operation	Header	Data	Data bytes
write	0x00	<ddt entry data>	4
map	0x01	<dst> <cap field>	2
grant	0x02	<cap field>	2
append	0x03	<order>	1
shrink	0x04	<direc><order>	1

The MPU ensures that the active SA has the permission to perform the operation on the DDT entry and in case of a map operation that the destination DDT entry is free.

The write operation is used to load a DDT during the boot-strap process, as the operation is allowed only when the MPU is disabled. The MPU expects the new data of the DDT without any checks. When the MPU is enabled the map operation has to be used instead. The map operation requires a new DDT entry, where the map is stored. In difference to the proposed map operation in Section 4.1.2 our current hardware implementation does not support a modification of the segment boundaries during the map operation. The software has to emulate this operation by using the append and shrink operations.

5.1.2.6 DDT entry look-up

We introduced two alternative implementations for a DDT entry look-up in Section 4.1.1.3. In case of implementing an internal memory for the DDT a CAM can be used. A DLB-based implementation can be used with internal memory as well as the external memory, but requires a stall signal due to the sequential DDT entry look-up.

CAM-based DDT look-up engine

A CAM requires that all memory addresses can be accessed in parallel, which is not possible in case of using a bus-connected memory. An implementation of a CAM based DDT entry look-up is shown in Listing 5.1. For each CAM address, 2^n downto 0, a separate match is generated. In case of a hit the corresponding `camIdx` line is raised.

Listing 5.1: CAM-based DDT entry look-up implemented in VHDL.

```
camMatch : for i in ((2**n) - 1) downto 0 generate
begin
    camIdx(i) <= '1' when rst_n = '1' and
                          (memAddr and ddtSizeMsk(i)) = ddtAddr(i) else
                 '0';
end generate;
```

For a determination of the permission rights in a subsequential match the current SAID is compared with the owner and the capability SAID field of the DDT entry addressed by `camIdx(i)`. In case of multiple matches the capabilities are combined to a single permission mask. Figure 5.8 shows a detail of the netlist of the subsequential SAID match.

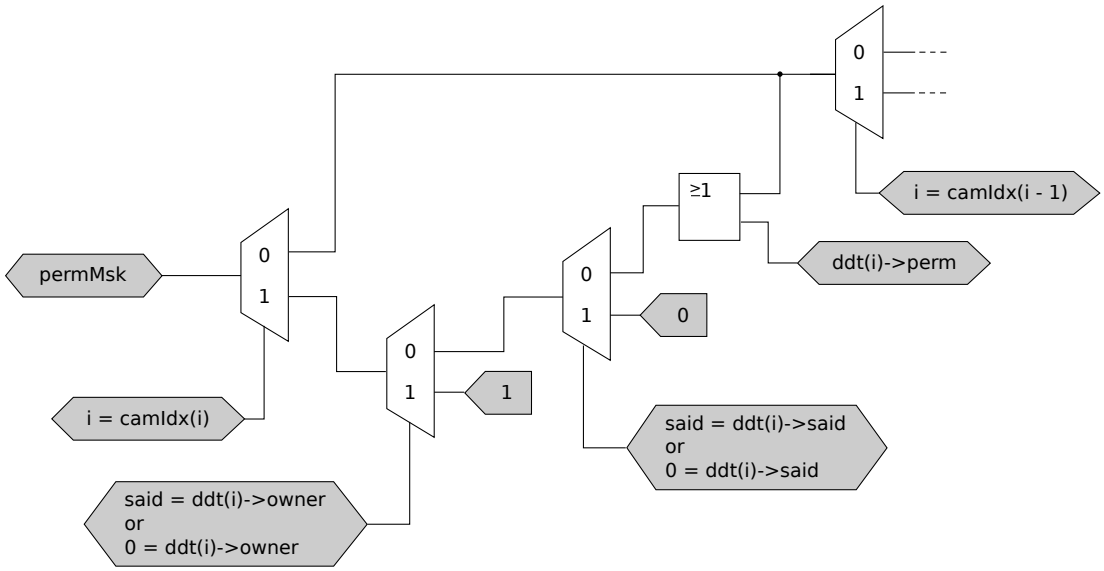

Fig. 5.8: A detail of the netlist of the subsequential SAID match.

Although a CAM-based DDT entry look-up can be performed in a single clock cycle, so that a stall signal is not necessary, its implementation as illustrated by Listing 5.1 and Figure 5.8 results in a very large design on an FPGA. We will give detailed information about the design footprint in Section 7.2.1. A more efficient implementation is possible by using a reconfigurable CAM as given by Guccione et al. [GLD00].

DLB-based look-up engine

We introduced the DLB-based look-up engine as an alternative design to the resource-hungry CAM engine. The DLB-based design can be used to implement a resource-efficient DDT storage by using an external memory connected by a memory bus. The design features two DLBs that hold the last DDT entries, to allow a look-up in a single clock cycle in case of a hit.

Due to the sequential DDT entry look-up and the sole DLB entry, overlapping data spaces are not supported by the DLB-based look-up engine. Hence, the map operation cannot be implemented in a way that it creates an additional DDT with an overlapping subregion. Instead, we have to extend the map operation to split the region in multiple regions. In the following the subregion with the demanded size is shared between the owner and the foreign SA. But the capability field of our previously defined DDT entry does not support shared data spaces. Thus, we extended the capability field to support multiple SAs.

Since an all-embracing scheme such as an access matrix, as introduced in Section 3.1.1, would waste a lot of memory, an ACL or a capability list have to be implemented. Figure 5.9 illustrates the memory required for implementing a ACL entry and a capability.

An ACL entry consists of a 3-bit permission field and a 16-bit array that indicates the SAs that get the permission for the given data space. A capability includes the SAID and a permission

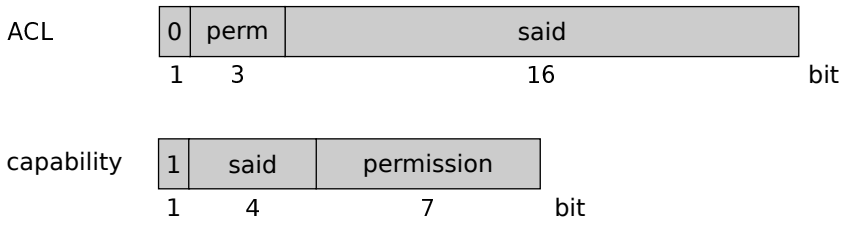

Fig. 5.9: ACL and capability list of a DDT entry used by the DLB-based look-up engine.

array assigned to it. As both schemes have advantages we decided to introduce a type indicator to make both types possible for a data space. The implementation of a shared memory where multiple SAs can get different permissions requires a list of elements. Due to the fixed size of a DDT element the size of capability field has to be defined at synthesis and depends on the memory spent for a DDT. But the size of the DDT is not as much restricted as for a CAM-based look-up engine.

The DLB look-up is a combinatorial logic element that matches the current address on the MAB with the DLB's content. A DDT look-up is enabled in case of a DLB miss. The current implementation uses a sequential search starting from the element zero up to the first element that triggers a hit. In case of a match the element is loaded into the corresponding DLB and the SAID match follows up. During the DDT look-up the processor's stall signal is raised, so that its operation stops until the look-up is finished.

5.1.3 Tiny hypervisor

We introduced a software-based isolation in Section 4.3.2. We figured out that a sandboxing approach cannot be used in combination with a dynamic memory management. Furthermore, we sketched that a full system emulation has an enormous performance drawback that would make the system unusable. Hence, we proposed a partial emulation that introduces a virtual instruction set (VIS) for memory access operations, function calls, and stack operations.

In the following, we illustrate an implementation of such a partial emulation in a tiny hypervisor on an MSP430 microcontroller. The tiny hypervisor is required on systems, which do not feature a hardware-based MPU as described previously. The tiny hypervisor replaces the hardware-based MPU and implements the primitives: *memory access control* and *data space management*. Further operations are implemented by the nucleus gate, which will be introduced afterwards.

5.1.3.1 Tiny hypervisor assembling

The primary target of the tiny hypervisor is to have control over each memory operation. Since a software-based mechanism cannot be placed physically between the processor and the system resources, we identified critical instructions that must be sandboxed by a trustworthy instance. Thus our tiny hypervisor must be invoked on each critical instruction. In contrast to the SFI approach we do not limit our approach on register-indirect memory accesses. Instead we emulate each instruction that is not a register-register operation.

The integration of the tiny hypervisor is done by a binary rewriter, as proposed by Wahbe et. al [WLAG93]. The binary rewriter replaces each critical instruction by a call into the tiny

hypervisor followed by a virtual instruction. The tiny hypervisor loads the virtual instruction from the program code just after its invocation. In the following, the memory address given by the virtual instruction is checked against the DDT. In case of a valid memory access the virtual instruction is performed and the hypervisor returns into the original program code, otherwise a trap is generated. Due to performance issues a function call and return are handled differently from the other virtual instructions in such a way that the PC is updated within the hypervisor and the program returns directly to the callee or to the caller.

The instruction emulation needs a temporary register. Thus, the compiler has to be parametrized to treat a register as a fixed register. The generated code will never refer to this register and the VIS as well as the tiny hypervisor can use it without restrictions. We analyzed in an example that the treat of a register increases the program code size slightly. Table 5.6 shows that the overhead for our Meetering app is 1.2 percent only.

Table 5.6: Comparison of the size of the Meetering app in case of treating two registers as a fixed registers (firmware compiled with gcc-4.4.5)

fixed registers	Text (bytes)	Data (bytes)	BSS (bytes)
-	28,522	738	1,277
R4, R5	28,868	738	1,277

5.1.3.2 Virtual instruction set (VIS)

We intended to realize the VIS as generic low-level language independent from the host ISA. But the ISA and the supported addressing modes determine the functional set of the VIS significantly. We chose the MSP430 MCU for our prototype implementation. As mentioned in Section 5.1.1.1, the MSP430 supports seven different addressing modes for the source and four addressing modes for the destination operand, which have to be considered by the VIS.

The VIS of the MSP430 tiny hypervisor includes six virtual instructions: *vload*, *vstore*, *vpush*, *vpop*, *vcall*, and *vret*. The *vload* operation loads the source operand into a fixed register. It uses the addressing mode specified by the original instruction. The *vstore* operation works similar for the destination operand. The *vload* operation and the *vstore* operation are always used in combination with the original instruction that is modified in a way that it uses the fixed register instead of the memory operand. The stack operations and the function calls are replaced by emulated instructions. Table 5.7 lists all instructions of the VIS for an MSP430.

Each emulated instruction is extended by a trap into the tiny hypervisor. We use a generic entry *__hyperv_entry* for all instructions except the *vret* instruction. The *vret* instruction does not require an additional operand, which has to be stored in the following virtual instruction. To omit the additional instruction we use a dedicated hypervisor entry point *__hyperv_entry_ret* instead.

Due to the complex addressing mode of the MSP430, instructions can use source and destination operands stored in memory. For these instructions a combination of *vload* and *vstore* instructions is necessary to replace a single instruction. We will give a more detailed view in our evaluation, in Section 7.2.2.

The MSP430 supports jump instructions with a relative offset of maximal 512 words. Therefore, it must be ensured that a jump instruction stays within the protection domain of an SA.

Table 5.7: VIS of the tiny hypervisor, the vload and vstore operations require an additional register regX to save the operand.

Virtual instruction	Original code	Generate code
vload	<instr> <src>, <dst>	call &__hyperv_entry vload <src> <instr> regX, <reg>
vstore	<instr> <src>, <dst>	<instr> <dst>, regX call &__hyperv_entry vstore <dst>
vpush	push <reg>	call &__hyperv_entry vpush <reg>
vpop	pop <reg>	call &__hyperv_entry vpop <reg>
vcall	call <func>	call &__hyperv_entry vcall <func>
vret	ret	call &__hyperv_entry_ret

As the MSP430 supports jump instructions with an immediate offset only. It can be guaranteed by a static analysis that the destination is within the protection domain. Only the branch instruction that allows a direct write of the program counter must be forbidden.

5.1.3.3 Guest interface

The tiny hypervisor provides a guest interface similar to the MPU register interface mentioned before. In contrast to the hardware-based MPU, the registers of the tiny hypervisor are ordinary memory objects. The tiny hypervisor gets control on each memory access just before the data are written. Hence, it can perform additional operations in case of an access addresses an MPU interface object. Therefore, a special software-based MPU interface is not necessary.

The register layout of the guest interface of the tiny hypervisor can be implemented similar to the hardware-based MPU. Thus, the nucleus gate and the guest application do not need any adaptations. A DDT and an SAID stack base register are not necessary. The addresses are resolved at build-time before the firmware's deployment. We will discuss the DDT in detail in the following subsection.

5.1.3.4 DDT implementation

The software approach of the tiny hypervisor makes a discussion about the location of the DDT superfluous, it must be stored within the MCU's main memory. Any external memory will not provide an adequate performance on TSSs. The implementation of the DDT has a significant impact on the system's performance. The chosen data structure will always be a trade-off between the performance of a DDT entry look-up and the memory resource allocation. We already sketched a DLB-based approach in Section 4.3.2. In the following we will briefly discuss a software-based implementation of a DLB-based DDT entry look-up.

Guarded DDT

We already mentioned the impact of the DDT entry look-up scheme at the resource allocation. Due to the limited resources of TSSs the DDT should be stored in a compact data structure. We proposed the concept of a guarded DDT, which allows an efficient entry look-up, an efficient storage, and supports pages with a variable page size. The use of a DLB requires the definition of non-overlapping data spaces, which is also necessary in case of implementing a guarded DDT. Thus, both concepts define similar requirements. Therefore, we chose a binary tree to store the guarded DDT and a DLB to store the last entry. The nodes of the binary tree are addressed by the base address of the data space and the subtree selector is the MSB of the remaining address. An example tree of a guarded DDT is illustrated in Figure 5.10.

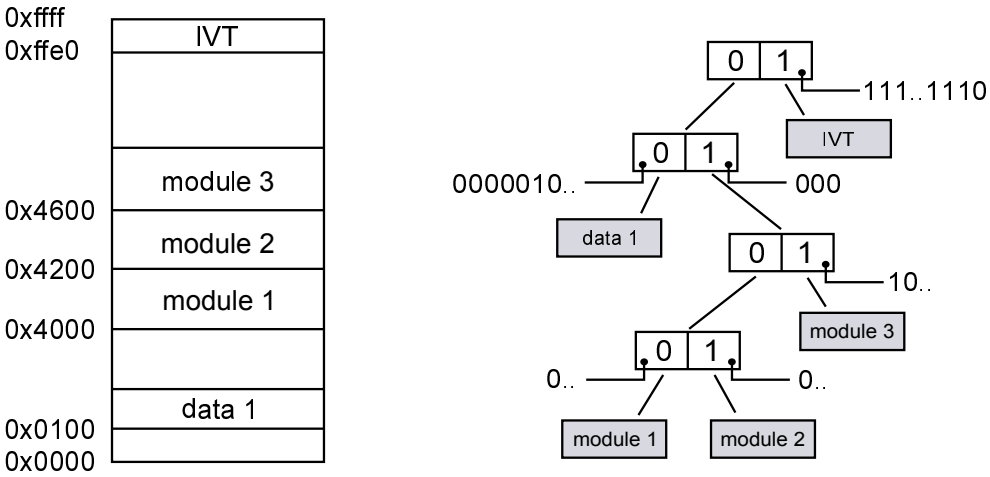

Fig. 5.10: The guarded DDT for an address space with five data spaces.

The data structure of an implementation of a guarded DDT as illustrated in Figure 5.10 is shown in Listing B.4 in Appendix B.2. We differentiate between nodes and leafs. A leaf contains the size, the owner, and the capabilities of the data space. The nodes are used to build the tree. Each node contains two elements, which reference the left '0' and the right '1' subtree. The nodes contain the node type, the guard length, and the guard. In case of a leaf a leaf object is referenced.

By using a guard, unused intermediate nodes are not necessary, so that the tree can be built in a very compressed form. It keeps the required memory and look-up operations small. Since each node object and each leaf object have the same size the memory management of the DDT is possible without fragmentation. The objects can be stored in a sole memory block, where the memory allocated for the nodes starts from the beginning and the memory allocated for the leafs starts from the end. The objects are linked by pointers, so that a run-time allocation and freeing is possible. The memory spent for the DDT can be defined application-specific as a configuration option.

Data space lookaside buffer (DLB)

Our hardware-based DLB approach proposes two DLBs, one for the data memory and one for the code memory. We followed this approach when assembling the tiny hypervisor as well. As we cannot check each memory access on an instruction fetch, we perform a memory

protection on data memory only. We mentioned in Section 4.3.2 that the code DLB is used on each call and return instruction. As function calls are frequently used, the additional resources spent for the implementation of an additional DLB will pay off in performance. In addition, a DLB selection is not necessary, it is given directly by the virtual instruction, so that any additional cost can be avoided.

The DLB was introduced to run a complex DDT look-up as rarely as possible. Although a DLB might avoid most of the complex DDT look-ups a DLB match is necessary on each memory check. Hence, the DLB match operation is of upmost importance. In hardware we implemented a combinatorial logic element that performs the match in parallel. In software a parallel match is not possible. The software match performance is rather determined by the width of the MCU's data path and the number of different instructions required to perform the match. Therefore, an implementation of a DLB match and the structure of the DLB must be optimized for the MCU's data path and the required instructions. The number of instructions can be reduced significantly by a pre-calculation of values such as the size mask. The ACL or the capability list must be transformed in a unified match optimized structure, so that additional conditions are unnecessary. The transformation will be performed at run-time just after a successful DDT look-up so that intermediate values calculated during the DDT entry look-up can be reused.

5.1.3.5 Run-time verifier

Since the tiny hypervisor cannot interrupt the guest application, it must be ensured that the binary was instrumented to redirect each critical operation to the tiny hypervisor. Similar to safety-related approaches a run-time verifier is strictly demanded.

In a security enhanced TSS we assume that the system has a boot-strap that executes a trustworthy instance before starting insecure operations. The trustworthy instance must invoke the run-time verifier that analyzes the program code at boot time. It has to check that all instructions use registers or immediate constants only. All memory access operations, stack operations, and function calls must be implemented by virtual instructions. Each call instruction remaining in the firmware must invoke one of the tiny hypervisor entry points.

As the run-time verifier must be part of the TSS application, its complexity must be quite low. The addressing mode and the length of an instruction are coded by the *Ad* and the *As* bits of the first instruction word and can be checked easily. The stack operations are emulated by move instructions, which can be checked easily as well. Function calls can be identified by checking the opcode. Therefore, the run-time verifier has to analyze the first instruction word only. All instructions that use a memory address must be encapsulated by the tiny hypervisor. The run-time verifier must be able to identify the tiny hypervisor.

5.2 The nucleus gate

The nucleus gate is the software-based part of the security nucleus. In common OSs the functions of the nucleus gate are usually part of the system kernel. We propose a kernel-less system and integrate a minimal set of functions into the nucleus gate. The nucleus gate implements functions that cannot be provided by hardware. It features:

- the static and dynamic DDT management,
- the CDC including access control, and
- interrupt handling.

We implemented security by the usage of isolated data spaces. Functionality, e.g. library functions, is located in a shared data space and can be used by public functions. Security critical data, e.g. the ACL, are stored in the private data space of SAs, so that a secure access can be guaranteed.

5.2.1 DDT management

The DDT management is an integral part of the nucleus gate. We differentiate between the static and the dynamic DDT management. The static DDT management includes all operations to prepare the initial configuration of the DDT. It includes the DDT setup at compile-time and the initial load during system's boot-strap. A dynamic DDT management becomes necessary when the MPU is enabled and any reconfigurations at run-time have to be performed. Since DDT operations have to use the limited MMIO interface, the number of required instructions to perform an operation may be significant. Hence, we propose a static DDT management that generates a DDT configuration that covers as much as possible run-time configurations. The dynamic management becomes necessary in case of supporting dynamic memory allocation or in particular to modify memory shares at run-time.

5.2.1.1 Static DDT management

The implementation of the static DDT management depends on the type of the DDT memory. In case of locating the DDT in the MCU's RAM the table can be built at compile-time as a static structure. The application bootstrap loader will automatically copy the DDT from the firmware image into the RAM just after system reset. Afterwards, the system has to store the DDT base address in the MPU DDT base address register and must only enable the MPU.

In case of using an MPU internal memory that can be accessed only via the MMIO interface, the initialization has to be done by a nucleus gate boot-up sequence. To reduce the overhead, the DDT memory can be initialized by a burst operation that copies a static setup of the DDT via the MMIO interface into the internal DDT memory. Hence, we introduced the DDT write operation.

Code segments

The static setup of the code data spaces is generated by the SPC based on the SPD. It groups software modules of one role in a single memory section that is assigned to a data space. In such a setup data spaces are owned by the SA that uses them first. Data spaces that are shared between SAs are assigned via delegation to other SAs.

In an ideal event-driven system the delegation of data spaces can be done dynamically at run-time. An SA that switches to another SA can delegate the needed data spaces first. But real systems are more complex, so that an SA usually does not own all data spaces needed

by its callee. Furthermore, a data space delegation is a complex run-time operation so that a static setup will be by far more efficient and should be preferred.

A data space, which is used by only two SAs, is initialized in such a way that it is owned by the first SA and statically granted to the second one. Hence, neither of them needs DDT management permissions on it. Data spaces, which are shared among more than two SAs, must be mapped to all of them, which requires a large number of DDT entries. We can reduce the number DDT entries in case of identifying SAs that are using the data space only once. In that case we can insert a dynamic mapping code section into the SA at compile-time that maps the code segments at run-time once after finishing its use. Afterwards, the SA does not need any permissions on the data space. We can identify such an SA at compile-time by analyzing the transition graph of the SPB. Each node of the graph, which is not part of a cycle, can give up its data spaces just before switching to next SA.

Data segments

Each software module defining global variables requires a data segment. Each global variable referencing to an object that is annotated to be public must be isolated from private variables. Hence, we need a private data space that holds all the private variables of a module. Furthermore, we need an additional data space that holds all public variables of a module. Since public variables are shared among multiple SA, they must be located in isolated data spaces. Later on, we will show that these data spaces can be assigned to roles, so that the number of data spaces can be reduced.

During compile-time the SPC identifies global variables and generates data spaces based on the SPD similar to code data spaces. In addition, data spaces for public modules are defined. These data spaces are assigned to the SA that uses those first. For SAs, which have not got a static grant or map of the data space, public data spaces are dynamically delegated to other SAs at run-time. The delegation instructions are added at compile-time by the SPC when generating the corresponding CDC.

5.2.1.2 Dynamic DDT management

Any DDT entry manipulation is possible under the restriction of the data space permission. Data spaces must be resized, created, mapped or granted by using the MMIO operations defined in Section 5.1.2. The nucleus gate extends the basic operations of the memory protection nucleus to provide a more comfortable interface to the DDT management operations.

Resize data spaces

Data spaces of TSSs are usually static. A resize operation as common on files in general purpose OSs is not necessary in TSSs. However, a resize operation is demanded to modify the memory layout of a TSS application at run-time. As mentioned before, we introduced an append and a shrink operation to resize a data space. We use these operations e.g. to map a limited view on a data space to a foreign SA or to create a new data space at run-time.

An SA that has the capability to resize a data space can expand or shrink the data space size. The resize operations are strictly coupled with the chosen data space description strategy. Our current implementation uses the SIT strategy. Therefore, the size can be increased or

reduced by an order of two only. The skrink operation can also be used to define a new base address of a data space. The operation is defined in a way that the MSB of the size parameter defines the position where the data space is shrinked. Nevertheless, the base address can be shrunk by an order of two. Hence, the definition of a specific base address and a specific size may require a multiple of resize operations.

The resize operations may be simplified in a significant manner in case of using an alternative strategy to describe the size of a data space. In case of using the EA or the SIB strategy the segment bounds can be described with byte-granularity. However, the used strategy is always a tradeoff between flexibility and performance.

Create data spaces

As already mentioned in Section 4.1.3 a create capability is not defined. Free memory is owned by the SAID zero, which gives all permissions to all SAs.[8]. A data space can be created by an SA by mapping a free section to its own protection domain by changing the owner field to its own SAID. Afterwards, each operation performed by any other SAID is forbidden on this data space unless it gets the grant permission on it.

As the dynamic creation of data spaces is rather uncommon in TSSs we did not optimize it in hardware. Instead the creation of a data space is a complex operation build on basic map and resize operations. Figure 5.11 illustrates the creation of a data space to access the timer registers of an MSP430. The registers are located at the addresses 0x160 up to 0x178. Due to the SIT strategy the data space size is set to 32 and includes the memory region from 0x160 up to 0x17f[9]. The assignment of the final size requires five resize operations. In a first step the data space is expanded to an address just after the target memory region. By using shrink operations the bounds of the data space can be moved to reduce the data space to the demanded size.

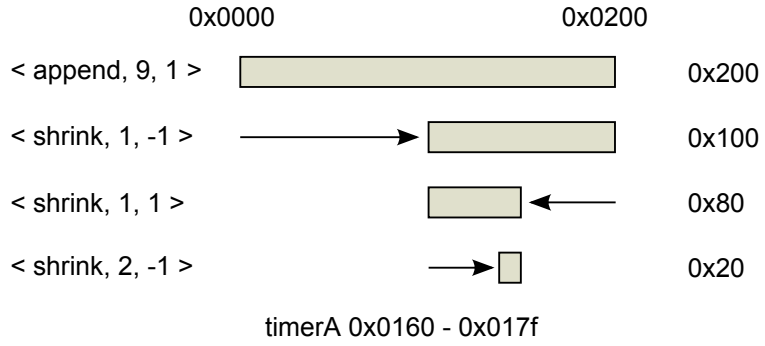

Fig. 5.11: The creation of a new data space for accessing the timerA registers of an MSP430 at address 0x160 - 0x178. It must be implemented as a complex operation build on append and shrink operations described by the triple $< operation, order, direction >$.

The tinyVLIW8 soft-core processor simplifies the scheme, it assigns each peripheral a base address with an order of two and the MMIO area of each peripheral is limited to an order of two as well.

[8]The ”zero” protection domain, which owns free memory, is similar to the sigma0 task of the L^4 μ-kernel run-time environment.

[9]Although the data space for accessing the timer registers is larger as required, the additional enclosed addresses are unused and can be assigned to the data space without any negative consequences.

Map data spaces

The map operation makes the allocation and the delegation of data spaces possible. An SA can map a data space if it is free, it is its owner, or it has the map capability to do it. We have described in the previous section how the map operation can be used to allocated a new data space.

Besides data space allocation the map operation is used to instantiate shared memory regions. In case of using a DDT look-up engine, which supports overlapping data spaces, the map operation makes a copy of the current entry and modifies the owner, the capability field, and the boundaries accordingly. In case of using the DLB-based DDT it must be differentiate between mapping a subregion or not. Since the DLB-based DDT does not support overlapping regions the data space has to be split for mapping a subregion. The subregions that describes the shared region is mapped to the foreign SA. The other subregion created by the split remain at the owner. A complete data space of a DLB-based DDT can be mapped by modifying the capability field.

Since the current hardware-based memory protection nucleus supports a very simple map operation only, a mapping as described above must be built by the nucleus gate in software. The software supported map operation includes the following steps:

- map the data space to itself to create a copy,
- resize the data space to the demanded size, and
- map the data space to the foreign SA.

Depending on the DDT look-up engine the first map and the resize operations result in a set of data spaces to ensure that the final mapping does not have any overlapping memory regions.

Grant data spaces

A data space is granted by modifying the SAID of the capability field. An SA can execute the operation if it is the data space owner, the data space is free or the capability field grants it the capability to do it. The grant operation is fully implemented by the hardware-based memory protection nucleus. Hence, the nucleus gate does not add any software support to the grant operation.

Revoke data spaces

A data space can be revoked from an SA by using a flush operation. The flush operation can be executed by the data space owner or by an SA, which has the grant capability. A data space is flushed by setting the SAID of the capability field to the owner's SAID. In case of freeing a data space the owner and the SAID field has to be set to zero.

5.2.2 CDC implementation

We introduced CDCs as a replacement of function calls in the original program code. The sequence of a secure CDC with eight steps was described in Section 4.2.1. Depending on

the type of the CDC, a complex parameter marshalling might be necessary. Nevertheless, each CDC can be implementated as a generic wrap-function. The generic CDC in pseudo code is shown in Listing 5.2.

Listing 5.2: The generated CDC saves the callers registers, grants parameters and initiates the context switch. Function-specific operations are addressed by the given function code.

```
PROCEDURE __wrap_f(said, fc)
        saveRegsFromCaller
        grantParametersUp(numParm[fc])
        clearRegsFromCaller

        switchContext(said)

        checkAcl(acl[fc])
        restoreRegsFromCallee
        call(f[fc])
END
```

The generic CDC is implemented as part of the nucleus gate. The SPC replaces the original program calls by the `__wrap_f()` function. To avoid doubled program code all CDCs variants are implemented by a sole wrap-function. The function gets the SAID and a function code (FC). The FC points into a CDC array that includes the function-specific information as number, type, and size of the parameters, and the address of the original function. The CDC array is generated by the SPC and placed in the memory as read only. We use a similar array to store the ACL of a CDC. Both arrays have to be separated since CDC array access and ACL array access take place in different protection domains. The compile-time generation makes a very compact storage of the entries of both arrays possible. The parameter information and the ACL of a CDC are stored in an array element with a variable size. Hence, an additional array is necessary to hold the start address of a CDC or an ACL element. Figure 5.12 shows a detail of a CDC array.

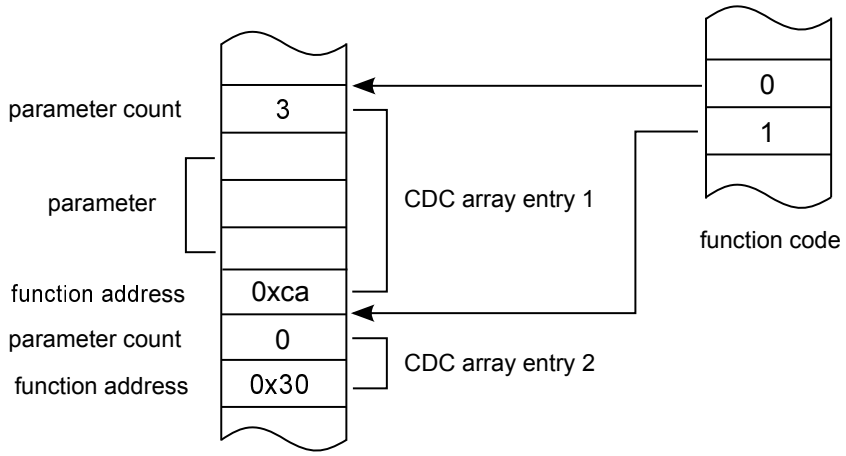

Fig. 5.12: The CDC array holds the function-specific information and is generated at compile-time. Due to its variable size the addresses to the array entries must be hold in a separate function code table.

We proposed a very fast domain switch by writing the new SAID into the `SAID` register of the MPU. Due to the fact that the following instruction is executed within the context of the new SA, the data space has to be changed within the domain switch operation as well. Changing the data space just after the SAID update would result in a very complex memory layout. By

using the generic wrap-function located in a shared code section, the CDC can be simplified in a significant manner.

Beside the generic CDC described above, a trampoline CDC has been introduced. Since non-trustworthy SAs may not have permissions to perform map or grant operations, a trampoline CDC is required to wrap these CDCs and perform an additional checks. A trampoline CDC is owned by a trustworthy SA, which have all permissions to perform grant and map operations. Furthermore, the trustworthy SA includes an ACL to check trampoline CDCs. After an successful access check the trampoline forwards the CDC form the non-trustworthy SA to the target SA and performs restricted operations as a proxy.

5.2.3 Access control

We already introduced the SPC of our extended compilation model in Section 4.4.3. The SPC processes the SPB and the annotated sources and generates native sources and the memory layout definition. In addition, the role-based ACL is generated and integrated into the native sources. We mentioned that the ACL is generated based on a CFG built at compile-time.

We differentiate between access control for functions and data segments. Data segments are always checked by the memory protection nucleus, so that an additional access control functionality is not necessary. An ACL is generated to control access on function level within a CDC only. Furthermore, rules are necessary to control access on the MPU registers. Since the registers are accessed by the security gate within the context of an SA, a fine-grained rule definition is necessary.

In the following, we introduce an MPU feature that is necessary to enforce a role-based access control on function level and an SPB extension that is necessary to prevent a privilege escalation by using the DDT management interface.

5.2.3.1 Role-based access control for CDCs

Oh et al. [OSM02] and Abadi et al. [ABEL05a] present a fine-grained control flow checking by testing jumps. The checks are enforced by a binary instrumentation at the jump destination (Oh et al.) or at the jump source (Abadi et al.). We introduced an adaptation of the RBAC model on TSSs, which allows us to define SAs and roles based on the application implementation. In Section 5.2.1 we mentioned that we map roles to data spaces that contain the role's software modules. By combining the current SAID and the currently used role, a control flow checking can be implemented that implements access control on function level. Due to roles define a set of modules our access control is more coarse-grain than the approaches of Oh et al. and Abadi et al. but supports a general access control instead of a binary instrumentation.

The role-based access control on CDCs requires an additional MPU register `DSID` that provides an identifier of the last recently used data space (DSID). As we are interested in data spaces that contain only code sections, a differentiation between code and data access is necessary. We already introduced an approach to support code and data DLBs that provides a differentiation[10], which can be reused to set the `DSID` register accordingly. On each mem-

[10] In case of using a CAM-based DDT entry look-up a differentiation based on memory addresses can be implemented.

ory access using a code data space the last data space ID is written into the `DSID` register. The IDs are defined by the SPC. On a CDC the SAID and the DSID are included into the permission check. We define: a function f can be executed by an SA $SAID$ in the role that executes program code of the data space $DSID$. Since the CFG and the data space IDs are defined at compile-time, an additional program code instrumentation is not necessary.

$$f \leftarrow \{USER(ROLE), \ldots\}$$

As an additional domain switch becomes necessary in case of performing a trampoline CDC, the callee SA, called by the trampoline, has no information about the caller SA, that invokes the trampoline. Therefore, a check of the access rights is not possible within the callee. Hence, the callee SA includes an access control that permits any accesses from the trampoline SA. In addition, it denies any access from any other SA. The actual access control is performed by the trustworthy SA, which has to check that the caller is allowed to invoke the callee.

5.2.3.2 DDT management access control

We integrated the MPU as an additional peripheral unit with an MMIO interface. The interface includes registers for controlling the MPU, for domain switches, and for the DDT management. The DDT management interface allows an SA the modification of data spaces that it owns or on which it has appropriated permissions. Furthermore, unused data spaces can be allocated.

DDT operation can be used to compromise the system, so an additional protection is necessary. The SPB is extended to include a flag that indicates untrustworthy SAs. These SAs do not get write access to the DDT management registers and must use a trampoline CDC to invoke functions of foreign data spaces.

5.2.4 Interrupt handling

The interrupt handling scheme of soft-core processors requires that the memory region holding the IVT must be readable by any SA. Therefore, we introduced an additional memory region that can be used by all SAs. Furthermore, trap code is inserted to enter the SA that holds the ISRs. We simplified the memory layout by using a single memory region that implements a generic ISR, which includes CDCs to the original ISRs. Figure 5.13 illustrates the interrupt handling on the secured platform.

In case of using an isolated SA for implementing an ISR bottom halve[11], as done by the radio SA of the Meetering app, an additional CDC is required to switch to the final SA. As an additional SA increases interrupt latency, so it must be used carefully. In our example applications, we moved the radio interrupt service to an additional SA as it processes possible malicious data. Reasoned by the fact that an ISR in a shared segment might have access to security-relevant system resources that can be manipulated by an adversary, such an isolation is strictly demanded.

[11] We split the ISR in a top halve and a bottom halve. The top halve is executed within the interrupt context and includes critical sections only. All further operations are performed within the bottom halve, which is activated by the top halve.

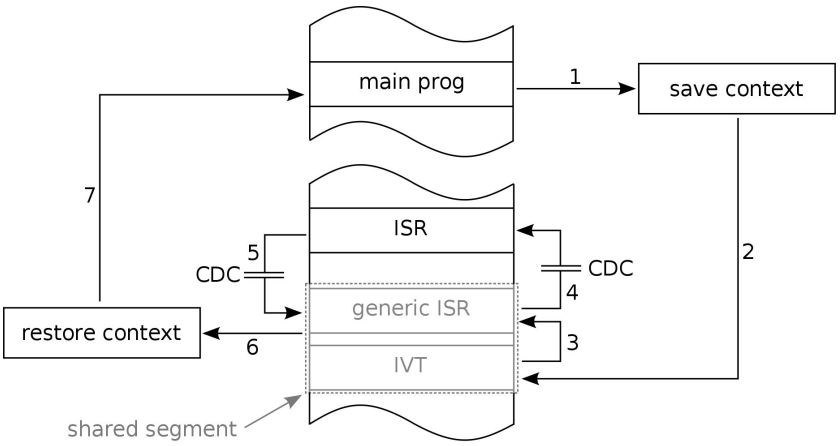

Fig. 5.13: Interrupt handling on security enhanced TSSs. The IVT and generic ISR code are placed in a shared data space. The original ISR is called by a CDC.

On an MSP430 the PC register and the status register are pushed on the stack of the current SA. The *return-from-interrupt (reti)* function restores both values. Therefore, it is required to return to the original SA where the registers were stored on the stack before executing the `reti` instruction. Therefore, the ISR must use a *CDC return* to return to the shared memory region.

CHAPTER 6

A secure platform of real tiny scale applications

In the last two chapters we introduced our platform for security enhanced TSSs and two possible assemblies of the security nucleus. In this chapter we will prove the applicability of our approach on real TSS applications. Hence, we describe a port of the two example applications on our secure platform.

The Meetering app has been written for langOS, which was also developed in parallel to the work on this thesis. We will introduce the OS and its compilation model first. Afterwards, we will explain the port of the Meetering application on a security enhanced MSP430 and the port of the SWUR application on the tinyVLIW8 soft-core processor that features a tailor-made MPU. Furthermore, we will describe an extension of the CoMet compiler suite, which is used to build tinyVLIW8 applications.

6.1 A security enhanced OS library for TSSs

We introduced a compilation model for our secure platform in Section 4.4.3. The model extends an OS library by the SPD and the SN library. We implemented the compilation model for the langOS OS library. In this section we will give a brief introduction to langOS and will explain the integration of the SN and the SPD.

6.1.1 An introduction to langOS

The development of langOs has been started during the IQlevel project. The project aimed to provide a wireless, low maintenance, digital and modular multi-sensor level probe. To cope with the flexibility of the ”Lego-like” sensor node IHPstack a high configurable sensor node operating system was required. langOS provides an easy to use and flexible configuration scheme and has a mandatory low power management that tends to enter the lowest possible power mode whenever no activity is recognized [SKK14]. The flexible configuration of langOS was driven by configuration capabilities of the IHPstack. The IHPstack was developed within the same project. An example configuration of the IHPstack with a power supply module, an MCU module, a radio module, and power amplifier module is shown in Figure 6.1 [SGG12].

The source code of langOS was published under the EUPL v1.1 free software license on the sourceforge online repository in June 2015 [Ste15b]. The OS is currently limited to the MSP430 MCU. It supports the TmoteSky, the IHPStack, and the IHPnode platform, which are mostly used within this thesis. Furthermore, the TmoteSky is a well-established mote in the community of wireless sensor networks and can be used with a broad variety of OSs.

Fig. 6.1: The IHPstack a "Lego-like" sensor node for low power sensor applications, the platform for langOS, a highly configurable sensor node OS library.

6.1.1.1 Compilation model of langOS

The OS sources of langOS are mostly written in C. The C programming language is widely used within the sensor network's community and gives a developer almost full control of the node's hardware. But for an implementation of the flexible configuration scheme of langOS the programming language C had to get a small extension. The required extension is realized by source code annotations, which are processed by a configuration compiler. The compilation model of langOS is illustrated in Figure 6.2. It includes an additional step to transform the extended sources into native sources. Afterwards, the native sources are compiled by a native compiler such as GCC.

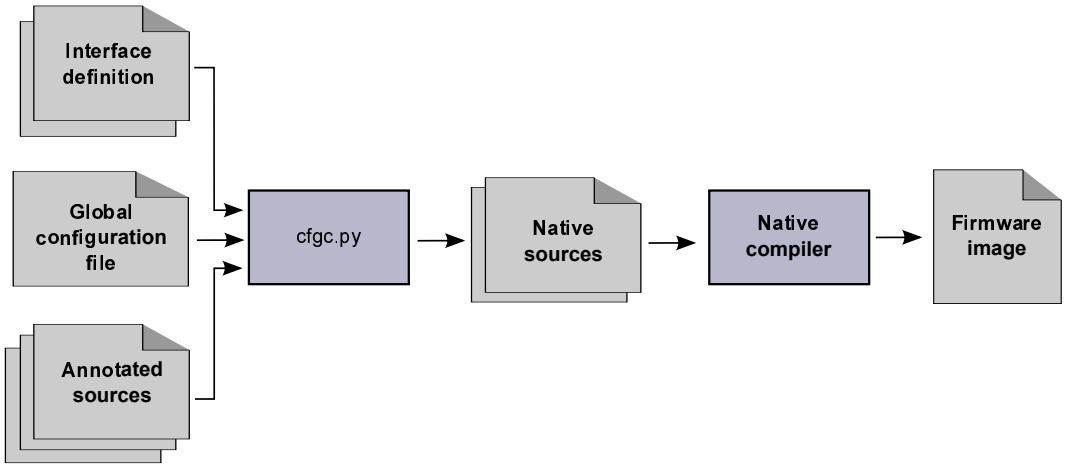

Fig. 6.2: The compilation model of langOS features a configuration compiler `cfgc.py` to compile annotated sources into standard C sources.

Each langOS application is separated in the langOS OS library and the application itself. These two parts are located in different directories, which simplifies the management of different applications. The application directory holds the application sources and the global configuration file (GCF). The GCF is used to control the build system by containing directives that define modules, which have to be integrated. The build process is fully controlled by the *make* build automation software. The build system traverses all directories of the application

and langOS in a recursive manner. In each directory all source files are compiled, if a corresponding directive is available in the GCF. The object files are archived to a single directory object file. In all upper directories the source files are compiled in the same manner and the subdirectory object files are archived to the directory object file.

The langOS sources are implemented by using a strict naming convention. The naming convention is needed by the configuration compiler to identify functions with special purpose and simplifies the understanding of the langOS sources. Each name of a function, a variable, or a macro that is private in the module starts with an underline. Public elements start with the name of the module including the path to the module, e.g the initialization function `init()` of the GPIO module `gpio.c` in the subdirectory `hal` has to have the name `hal_gpio_init()`.

6.1.1.2 Tailor-made configuration

The configuration compiler of langOS transforms source files with a retrofitted C programming language into native C sources. The extension is primarily used to solve module bindings and to integrate configuration attributes at compile-time. The process is very similar to the OSEK implementation language [Zah98] and consists of the three basic primitives: the global configuration file (GCF), interface descriptions, and annotated sources. In the following we will give a brief description of the primitives. Further details are given by the master thesis of A. Krumholz [Kru15].

Global configuration file (GCF)

The GCF of langOS was initially designed similar to the configuration file of the Linux kernel. The file includes directives for enabling modules and basic attributes for configuring the modules. It is included by the automatic build system and processed by the configuration compiler that generates a global header file, named `autoconf.h`. The global header file is included in the C programming sources to have access to the configuration attributes. The master's thesis of A. Krumholz extended the GCF to include also interface bindings and hooks [Kru15].

Interface bindings are used to assign software modules to an interface. Interfaces are an abstract representation of modules that are used within other modules. The real implementation of a module is solved by the assignment of a module to an interface within the GCF.

Hooks imitate the concept of advices of AOP used in the CiAO operating system. A hook is a function that can be used to extend sources without modifying the source itself. It can be assigned to a function as well as to a module and is executed either before the assigned function is called or after its invocation. In case of an assignment to a module the hook is called if any function of the module is invoked.

Attributes in the GCF are always associated with a module. The definition of these attributes within the GCF is mandatory if the module is activated by the configuration. The configuration compiler converts attributes in macros, which can be used within module sources.

The modularity of langOS makes the implementation of functional alternatives possible. Therefore, we introduced the concept of an interface, which describes an abstract signature of modules. The signature includes all public functions and attributes, which must be implemented by a module. A module is assigned to an interface by the GCF. Furthermore, modules use interfaces instead of a concrete module function to make a functional alternative possible. The final binding is done by the configuration compiler.

The definition of an interface can be extended by inheritance. An interface that inherits another interface can add further functions and attributes. But polymorphism and multiple inheritance are not allowed.

Interfaces are defined in separated interface definition files. These files are located in the application as well as in the langOS global interface directory, which simplifies the look-up of an interface file by the developer and by the configuration compiler.

Source code annotations

Since all langOS sources are implemented in the C programming language, the integration of interfaces was not possible without a small extension. Therefore, we introduced the delimiter `::` to be use with an interface member, where an interface member is either a module function or a module attribute. By the delimiter the configuration compiler is able to identify an interface member by the following abstract signature

`<interface>::<member>`.

The configuration compiler of langOS parses the source file for the delimiter `::` and replaces the abstract function call by the correct function name. Furthermore, it includes the required header file, so that signature errors can be identified by the native compiler.

6.1.1.3 Boot-strap and main-loop

The boot-strap of langOS is divided in three steps: the module initialization, the run-once tasklets, and the OS main loop. The three steps are controlled by the C main function, which is called by the boot-strap code. The boot-strap code is generated by the C compiler and ensures that the data section is initialized before calling any C function.

The highly configurable module selection of langOS asks for an automated module initialization scheme. Hence, the langOS configuration compiler generates an initialization array, which is traversed by the langOS boot-strap. Each module that defines a function with the signature `int <module_name>_init(void)` will be initialized automatically. Due to the fact that the order of the invocation of the module initialization functions is unpredictable, functional blocks can be moved into run-once tasklets. Run-once tasklets are started just before the OS main loop is entered. The main loop invokes the langOS task functions[1] by a simple round-robin scheduler.

[1] A langOS task function is a coroutine as defined by Melvin Conway [Con63].

6.1.2 Constructing data spaces

The enforcement of tailor-made data spaces is a basic primitive of the platform for security enhanced TSSs. Hence, in a first step we will present how data spaces can be constructed for a langOS application. We mentioned that a data space includes program code, program data, and peripheral resources of a TSS. The construction is based on grouping program modules according to the SPD and source code annotations. Since each langOS application build is based on a tailor-made configuration that controls the build process the construction of data spaces is done analogously to the GCF.

The construction of data spaces includes three steps: an extension of the langOS compilation model, the consecutive grouping of program sections, and the initialization of data spaces at run-time.

6.1.2.1 Extended langOS compilation model

We have introduced the langOS compilation model in Section 6.1.1. For the construction of data spaces the model has to be changed in such a manner that all object files are linked in a single step. The langOS compilation model that builds archives for each subdirectory recursively is not suitable. The building of archives prevents a grouping of program sections of modules from different directories.

The new compilation model uses the configuration compiler to generate native C sources and collects these files in a single directory. Similar to the naming conventions enforced within the langOS sources the unique file names are constructed by prefixing the path within langOS separated by an underline. The configuration compiler that processes the GCF `app.conf` and the langOS library file `dev/uart.c` generates the output file `oslib.dev_uart.app.c`. The prefix `oslib` is used to generate a unique file name. Files of the application directory get the prefix `app` to differentiate them from langOS library files.

The generated files are compiled by the native compiler and linked to the final firmware file. The linking process can be extended to perform a grouping of program sections.

6.1.2.2 Consecutive grouping of program sections

Each object file generated by a native compiler consists of at least three different sections: a `text` section for the executable program code, a `data` section for initialized data, and a `bss` section for non-initialized data. For an enforcement of protection domains, sections of the same type of different object files have to be grouped consecutively. Furthermore, labels for the start address and end address of a data space have to be defined. Listing 6.1 shows the linker script of the GNU linker `msp430-ld` to generate data spaces.

The `MEMORY` command is used to define global regions of the address space. The regions are based on the physical memory of the MCU and may differ for different MCUs. The `SECTIONS` command is used to place object file sections into the global memory regions. The command allows a customized ordering of object files. We define start and end labels to delimit the data spaces. The labels can be used later to refer to the data space bounds. The grouping of object files is based on the SPD. Hence, the linker script has to be generated by the SPC and is always MCU and SPD specific.

Listing 6.1: MSP430 `gcc` linker script to generate data spaces by grouping object file sections. Based on the result of the Master's thesis of E. Bergmann [Ber12].

```
[...]
MEMORY
{
  text    (rx)    : ORIGIN = 0x4000,     LENGTH = 0xbf5f
  data    (rwx)   : ORIGIN = 0x1100,     LENGTH = 0x2800
  vectors (rw)    : ORIGIN = 0xffe0,     LENGTH = 32
}
SECTIONS
{
  .text :
  {
    [...]
    _pd1_TextStart = .;
    D1_a.o(text)
    D1_b.o(text)
    [...]
    _pd1_TextEnd = .;
    _pd2_TextStart = .;
    D2_d.o(text)
    [...]
    _pd2_TextEnd = .;
    [...]
  } > text
  [...]
}
```

Similar to the `text` section the `data` section has to be grouped. But for the creation of a consecutive data section the `bss` section must be included as well. The standard linker script arranges the `bss` sections of all object files behind the `data` sections. However, we can integrate the `bss` section into the `data` section by adding some modifications to the boot-strap code and the linker script. But it is much simpler to initialize all variables, so that they are always placed within the `data` section by the compiler.

Whereas public functions are accessed by a CDC and no differentiation between private and public functions is necessary, public variables must be handled differently. Since public variables are annotated in the source code, their placement in a public section can be enforced. We make use of the GCC attributes to implement such a placing. Afterwards, public variables are processed by the linker script similar to other sections.

6.1.2.3 Data space initialization

It is mandatory that the DDT initialization is performed by a boot-strap module that has read access to the start and the end label of the data spaces. Depending on the chosen MPU implementation the DDT is either constructed at compile-time and copied into the system memory at boot-time or transferred into the MPU memory by using the MPU write command.

Since each langOS application starts with a fixed boot-strap the initialization of the DDT is added to the OS library. Since the DDT implementation is part of the security nucleus, it is implemented as a module of langOS and can be activated in the GCF in combination with

the SN. The DDT implementation makes use of functions of the nucleus gate, which covers also the differentiation between a hardware-based and a software-based MPU.

The boot-strap of langOS guarantees that each application starts with the main function of the OS library. The main function invokes the application initialization function after initializing all modules. Hence, it is mandatory to start the DDT initialization at the beginning of the main function or just before it. We use the following a function attribute to place the DDT initialization function in a program section that is executed before the main function is called:

```
void __attribute__ ((section(".init6"))) ddt_init(void)
```

The use of an attribute is less invasive than changing the main function and ensures that the initialization is finished before any langOS library function is called.

6.1.3 The SN integration

We introduced the SN and its basic primitives in Section 4.3. Chapter 5 has given a description of the assembling of the SN on two real ISAs. The langOS OS library is focused on the MSP430 platform, which is one of the afore mentioned real ISAs. In the following, we describe the integration of the basic primitives of the SN into langOS. Figure 6.3 illustrates the components of the SN with a hardware-based and a software-based MPU driver.

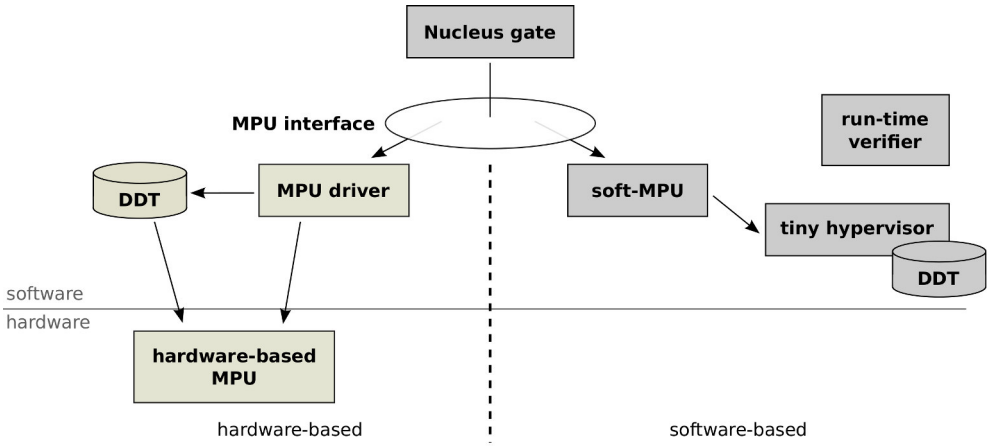

Fig. 6.3: The langOS library supports a software-based as well as a hardware-based SN implementation. A unique application interface is provided by the nucleus gate, which uses the MPU interface to access the instantiated implementation.

We make use of the langOS interface bindings to provide a sole MPU interface to the OS library and the applications, irrespective of using a hardware-based or a software-based MPU implementation. The interface is bound to the MPU implementation based on the GCF. In case of configuring the software-based MPU, the libraries of the tiny hypervisor and the runtime verifier are automatically added to the firmware.

6.1.3.1 Nucleus gate

The nucleus gate has been implemented for langOS as an additional module that can be enabled by the GCF. The module includes the data space management and the cross domain

communication primitive. Although, the implementation might be independent from the OS, we have chosen a tightly coupled implementation to make use of langOS specific features and to keep the nucleus gate memory footprint small.

Cross domain calls

The initial implementation of langOS makes an extensive use of function-pointers to implement configurable bindings between modules. The master's thesis of A. Krumholz has replaced the dynamic binding of langOS by a compile-time approach. Hence, the langOS library is implemented mostly without function-pointers. Especially an interface that defines an abstract binding between modules works without function-pointers now.

We make use of the interface replacement to integrate CDCs. To avoid unnecessary CDCs, the SPC identifies modules, which have to provide a public interface. The configuration compiler of langOS processes interface files to generate dictionaries, which include attributes, interfaces, and hooks. These dictionaries are used during the compilation process. We extended the dictionaries to include also CDC information. This information is used to integrate domain switches as described in Section 4.2.1.

MPU interface

The MPU interface decouples the used MPU implementation from the nucleus gate implementation. The Listing B.1 in Appendix B.1 shows the langOS interface that has to be implemented by the soft-MPU and the hardware-based MPU driver.

MPU driver The MPU driver provides an API for the MPU registers. The registers and flags are defined in a system header similar to peripheral units. The header is given in Listing B.2 in Appendix B.1. In case of using an external DDT the MPU driver allocates the memory as well.

soft-MPU The software-based MPU includes a virtual MPU register interface, where each register is implemented by a local variable. The register addresses are given to the tiny hypervisor that implements the DDT memory. In case of a read or write access the tiny hypervisor can redirect the operation to its internal data structures.

The MPU interface provides also functions for a dynamic data space management. The functions are mostly used by the nucleus gate for data space delegations during CDCs. Although, the langOS library provides a dynamic memory management based on the buddy system algorithm, it does not use it, so that dynamic allocated memory cannot be used across domain applications yet. Since performance issues the data space initialization during the system's boot-strap uses native operations, which are not part of the interface, e.g. the write command.

6.1.3.2 Tiny hypervisor

The use of a software-based memory protection is tightly coupled with the tiny hypervisor. Its integration is activated by the GCF and requires some further modifications on the langOS

build chain. The tiny hypervisor was implemented as a software library that is linked to the langOS application in the final build step. We implemented the hypervisor in such a manner that it does not have any dependencies to the langOS OS library. The tiny hypervisor features the VIS emulation, the memory access check, and the DDT management.

Instruction emulation

The VIS emulation is partially implemented in an assembler language. As shown in Figure 6.4, the emulation of a virtual instruction consists of three steps: instruction fetch, access control, and instruction execution. We decided to implement the instruction fetch in the assembler language. The assembler language gives us control on the used registers. Due to the fact that a virtual instruction may use any register, the instruction fetch has to ensure to use free registers only. Otherwise it cannot be guaranteed that the register content is not overwritten. We sketched already the need of temporary registers for a successful integration of the tiny hypervisor. We make use of these registers during instruction emulations. It ensures that we have access to the original register content and do not need any additional stack operations that modify the application's stack layout.

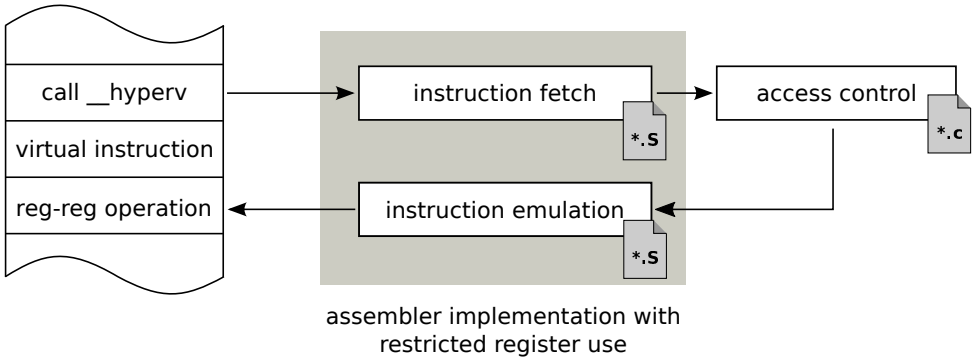

Fig. 6.4: The instruction emulation on an MSP430 is split in three steps: instruction fetch, access control check, and instruction emulation. The instruction fetch and the instruction emulation must be implemented in assembler (.S) to ensure that application registers are not overwritten. The access control can be implemented in standard C (*.c).*

The access control is implemented in the C language. Given that the instruction fetch has saved the register content the access control has no register restrictions. It simplifies the implementation of the access control significantly. The instruction execution is implemented in the assembler language again. It must be ensured that the register content is prepared properly before a return to the normal program is executed.

To simplify and to optimize the run-time performance of the tiny hypervisor, it does not perform any sanity checks on the instruction emulation. The correct generation and the verification of the VIS is part of the binary re-writer and the binary verifier. They ensure that a native instruction uses registers only. Any memory access must be performed by a virtual instruction.

Binary re-writer

The binary re-writer was introduced to close the gap between the native OS implementation and the tiny hypervisor. It instruments the TSS application to ensure that the tiny hypervisor gets control on each memory access. It is mandatory that the binary re-writer has control on

the final firmware image. Hence, the integration must be done as close as possible to the final compilation steps to ensure that no additional memory access is integrated.

We modified the native compilation step of the langOS build chain in such a manner that we generate an assembly file first. The hypervisor instrumentation works on the assembly file and transforms it into an instrumented version. In a following step the file can be compiled to an object file that is linked to the firmware image. Figure 6.5 illustrates the extended compilation process.

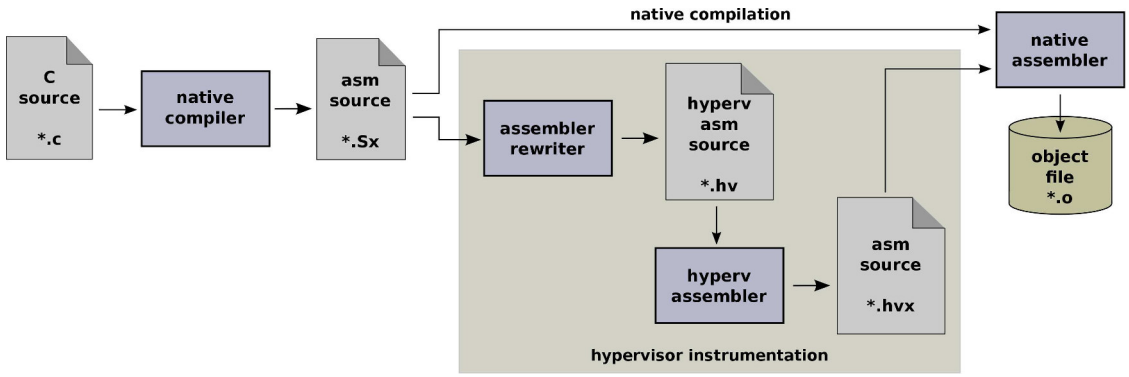

Fig. 6.5: The langOS build chain was extended by an assembler re-writer and a "hyperv" assembler to instrument assembly sources before generating the final object file.

Our approach does not support a binary rewrite at run-time as presented by the t-kernel [GS06], Harbor [RKS07], or SFI [WLAG93] and requires the program sources. But offline instrumentation reduces the size of the firmware image and makes expensive optimizations possible. We give a comparison of the design size of the different approaches in Section 7.2.1.

Notwithstanding our previously introduced approach in Section 5.1.3, we added an additional transformation that converts instrumented assembler code into native assembler code. We introduce the *hyperv assembler*, which is much simpler than an assembling tool that generates a binary file. In addition to a simpler assembly tool, we can make use of native instructions in such a manner that we can execute the native assembly code unchanged on the host's processor. The tiny hypervisor has not to emulate these instructions, which avoids the performance drawback of the instruction emulation. Table 6.1 shows the transformation of the virtual instructions into native instructions.

Table 6.1: Mapping of virtual instruction to real instruction to make use of a native assembler.

virtual instruction	native instruction	executed
vload <op>	mov r2, <op>	no
vstore <op>	mov r3, <op>	no
vpush <reg>	push <reg>	yes
vpop <reg>	mov @r1+, <reg>	no
vcall <op>	call <op>	yes

The use of native instructions complicates the run-time verification. Since the replacement might be part of ordinary program sections, the binary verification at run-time cannot use a special instruction opcode. It has to identify the virtual instructions using the hypervisor entry point instead. But the identification is quite simple because of the hypervisor entry point is a

call instruction with an absolute addressing mode, which makes it unique within the overall firmware image. Hence, the use of native instructions for the virtual instructions instead of special instructions has no drawbacks on the binary verifier step and is more a performance optimization at run-time than a security means.

Run-time verifier

The binary re-writer is implemented as a desktop application and integrated in the langOS compilation model. Its integration ensures that the final binary is instrumented after compilation. But the model cannot guarantee that the binary is not manipulated later. Hence, a run-time verifier is necessary to ensure that the tiny hypervisor is integrated as described in Section 5.1.3 and the following items are fulfilled:

- all instructions use either the register mode or are virtual,
- for all `vstore` operations the AD-bit[2] is set and the AS-bits are cleared,
- virtual instructions do not use the symbolic addressing mode, and
- the program counter (R0) is never the destination of a `mov` instruction.

Due to the fact that langOS does not support loadable program sections, a run-time verification needs to be executed only once. We extended the langOS boot-strap similar to the DDT initialization to integrate the program code verification before calling any langOS OS library function. The verifier uses the text-start and text-end labels introduced by the linker to identify code sections. We assume that all data between these labels are program code and can be verified by reading the first instruction word to identify the instruction, the addressing mode, and the instruction size.

The system start is delayed by the run-time verifier and in case of a verification error the boot process is aborted[3]. Any insecure operation must be started afterwards.

6.1.4 RBAC on langOS

Section 4.4 has presented an adaptation of the terms of the RBAC model for TSSs. To provide a RBAC model we extended each langOS application by source-code annotations and a text file that contains the SPB.

6.1.4.1 The SPB of langOS applications

The SPB of a langOS application includes the definition of software activities, roles, role assignments and user transitions. We define the terms considering the langOS specifics as follows:

[2] The instruction word of a `mov` instruction that does not use a register target has set the AD-bit.

[3] A proper error handling is out of the scope of this thesis. A hard abort prevents harmful programs from the execution of malicious code sections. But a dependable application requires a more complex handling.

Software activity An SA in langOS is defined as the software module that includes the function that starts a langOS task.

Roles Due to the fact that a direct mapping of each module to a single role would result in a large number of roles and in addition roles may be assigned to multiple SAs an approach to group modules to form roles is demanded. We define a role as the maximal set of modules used only by the same set of SAs.

Role assignment Roles are assigned as a set of roles to an SA, where the set of roles includes all roles used by the same SA.

User transitions A user transition from an SA SA_i to SA SA_j is given if a function of a role assigned to the SA SA_i calls a function of a role assigned to the SA SA_j.

Due to the fact that SA are extended to the surrounding software module, a protection domain must be changed if any function of the software module is called. In a well-defined application the primary definition might be still retained or can be constructed by an additional trampoline function. But for performance reasons it is more efficient to provide additional public functions and switch the protection domain on a call of one of these functions.

6.1.4.2 Source-code annotations

Source-code annotations are used to identify public functions, variables, and MMIO sections. As already mentioned in the last section, we extended the interface definition of langOS to integrate CDCs definitions, so that no further source-code annotations are necessary. Hence, only the configuration compiler has to be qualified to process the extended syntax of interface files.

We described that public variables are identified by a C pre-processor macro. We use the gcc attribute to place each public variable in a section called `public`. The section is used by the linker to place all public variables in an isolated section. Based on the separation of private and public variables, we can form data spaces for all private and public variables and assign them to roles. The IO resources are defined within the hardware abstraction layer (HAL) driver modules of langOS. For each MMIO an isolated data space is created by the SPC. The data spaces are assigned to the roles defined in the SPB.

6.2 A security enhanced *Meetering* app

We introduced the *Meetering* app of the *Diamant* project in Section 2.5.1. In the following, we will describe the port of the application to the security enhanced platform for TSS presented in this thesis. The application was implemented for the TmoteSky [Cor06] as well as for the IHPnode [PSL10] platform. The functionality on both platforms is almost identical except that the TmoteSky supports a smaller number of lamp clusters and the radio driver uses an IEEE802.15.4 compliant network protocol.

We use the TmoteSky, which is fully supported by the MSPsim, to demonstrate our hardware-based security nucleus for the MSP430. Furthermore, we use the IHPnode to demonstrate our software-based approach. The IHPnode is equipped with an MSP430F5438A, which implements the MSP430X ISA and has 256 kB of non-volatile memory. The large amount of non-volatile memory simplifies the prototype implementation of our security nucleus in a significant manner.

As already mentioned the implementation of the Meetering app is based on the langOS OS library. Given that the application and the OS library were developed without our platform in mind, the software port as well as the SPD have to accept trade-offs. First, we were forced to add a few modifications to the original implementation. But we are convinced that all these modifications do not impact the functionality. Nevertheless, an enlargement of the program size cannot be avoided. Second, since langOS is focused on resource-efficiency and low latency, it does not enforce a software architecture with per se isolated modules. Hence, the SPD must be able to handle software modules with an extensive shared use.

6.2.1 Module separation

The Meetering app was initially implemented in a single software module. Although such an implementation is unusual in common software systems, a TSS application may include only few lines of code, which can be handled in a single module. Most functionality is provided by the langOS OS library. As our secure platform requires a separation of SAs or software components in different modules, we identified and isolated the software modules for the Meetering app first, see Table 6.2.

Table 6.2: Application modules of the Meetering app.

Module	Description
meetering	Applications boot-up module.
capctrl	Control module, that enables and disables the lamps.
network	Network packet handler
storage	Storage module, that writes the current module state into the flash memory

Beside the four modules of Table 6.2 the Meetering app uses 22 software modules of the langOS OS library. The complete list of all software modules of the Meetering app can be found in Appendix A.1.1.

6.2.2 RBAC for the Meetering app

Based on the separation of software modules a role-based access control can be defined for the Meetering app. We followed the scheme introduced in Section 6.1.4 and filled the SPB.

6.2.2.1 Security policy book

The initial description of the Meetering app in Section 2.5.1 appoints three basic activities. Based on the application of the security platform we can identify six SAs. These are capctrl,

network, and storage, which are similar to the modules of the Meetering app listed in Table 6.2, and in addition the SAs boot-strap, main, and radio.

Boot-strap Because of the automated module initialization scheme of langOS, the OS main function uses functions of nearly all modules. Hence, the boot-strap activity of langOS needs access to all regions, so that we were forced to define a separate boot-strap activity.

Main The main loop of the round robin scheduler is assigned to the main SA. The SA needs access to all modules that are not separated in an isolated SA and to all modules that define a suspend- and resume-function[4].

Radio The langOS network protocol stack is split in a bottom and a top half. The top half is executed within the interrupt service routine registered by the radio driver. The bottom half includes the more complex protocol handling. To separate both parts we defined an additional software activity that features the bottom half of the network stack.

An overview about the software modules of the Meetering app and their usage by SAs is given by Table A.1 in Appendix A.1.1.

Defining roles

We constructed the CFG of the Meetering app to define roles. The complete graph is shown in Figure A.1 in Appendix A.1.2. We use the semantics shown in Figure 6.6.

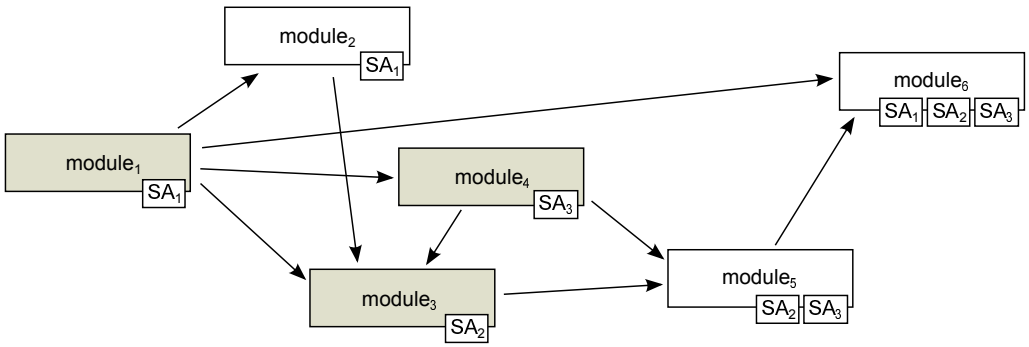

Fig. 6.6: Semantic illustration of a CFG of a TSS application to identify elements of the RBAC model.

Each module is labeled in the small box by the first character of the name of the SAs that use the module. A module that is assigned to an SA is shown in the gray box in the graph. Roles are defined as described in Section 6.1.4. We used the first character of the SAs to build the name of a role. Listing A.1 in Appendix A.1.2 shows the resulting fourteen roles and the role assignments of the Meetering app.

[4]The suspend and resume function can be defined by any module to be notified when the system enters and leaves a low power mode. More information regarding the suspend and resume of langOS is given by Stecklina et al. [SKK14].

The final number of needed data spaces depends on the implementation of the look-up engine. In case of using a rights lookaside buffer (RLB)-based look-up engine we need only fourteen data spaces. We can use an ACL in the DDT entry to assign the execution right to multiple SAs. In case of using a CAM-look-up, we need 30 DDT entries to map each role to a data space. We can reduce the number of roles by using the grant operation. Each module of the Meetering app, which is assigned to more than one SA, is initially used by the boot-strap SA. Therefore, data spaces can be owned by the boot-strap SA and mapped to further SAs later. Hence, the minimal number of data spaces to hold all code segments of the Meetering app is 22 in case of using a CAM.

Role transitions

The role transitions are derived from the directed edges of the CFG. Based on the CFG of the Meetering app we identified the transitions of Listing A.1 in Appendix A.1.2. The boot-strap SA initializes all software modules after power on reset. Hence, we have to allow a transition to each SA. In the following, the boot-strap SA is not used again, so that a transition from any SA to the boot-strap SA is not necessary.

Complex SAs as the main and the network SA need a wide access as well. The main SA implements interrupt handling and task scheduling and needs transitions to the radio, the capctrl, and the network SA. The network SA implements the langOS network protocol stack, which needs transitions to all SAs except the boot-strap SA. All other SAs need only few transition, because of they are mostly endpoints. Due to the fact that the number of transitions defines the number of CDCs, it has a direct impact on the system's security. We will analyze detailed the number of CDCs in Section 7.1.1.2 by examining the remaining computing base.

Define data spaces for data and MMIO sections

In 18 software modules of the Meetering app global variables are defined. Similar to the code sections, non-public variables are assigned to the 14 (RLB) or 22 (CAM) roles defined in the SPB. But for private variables we need one data space less than for the code segments given by the fact that one module defines only public variables. In addition, within the 18 modules 26 public objects are defined[5]. The SPC isolates the public variables in separate data spaces, so that we need additionally 18 data spaces.

Table 6.3 summarizes the data spaces defined for the Meetering app. We differentiate between a CAM-based and a RLB-based DDT implementation. Due to the fact that the CAM-based DDT supports one SA per DDT entry only, shared data spaces must be mapped to multiple DDT entries.

MMIO sections are used by the hardware abstraction layer (HAL) of langOS. The Meetering app uses eight HAL drivers: digitally controlled oscillator (DCO), GPIO, two Timers, Universal Asynchronous Receiver Transmitter (UART), SPI, FMC, and watchdog timer (WDT). Given that each MMIO section needs an isolated data space, eight additional data spaces are needed.

Since the DDT management interface is accessible by the MMIO interface, an isolated data space is needed to control access to the MPU registers. As radio driver and network stack

[5]Three public objects defined by modules of the Meetering app can be made private without consequences on the functionality in case of adding few modifications in two langOS modules.

Table 6.3: Data spaces defined for the Meetering app. Because of the different capabilities of the CAM-based and the DLB-based DDT, different numbers of data spaces are necessary.

	Number of data spaces	
	CAM-based	DLB-based
code segments	22	14
private variables	21	13
public variables	18	18
IOMEM	8	8
MPU MMIO	6	2

are vulnerable by external network packets, the radio SA and the network SA may not get any access to the `ADDR` and the `DATA` register of the MPU. Therefore, a sole data space is not sufficient and at least two different data spaces are needed. In case of a CAM-based DDT six data spaces are needed.

6.2.2.2 Access control list (ACL)

As mentioned in Section 6.1.4.1, we have simplified the definition of SAs. Therefore, a CDC is performed if any function of the module that defines the SA is called. Furthermore, we described the application of RBAC for CDCs in Section 5.2.3.1. On that account the ACL of a langOS SA must include sets of tuples $< SAID, DSID >$ for all public functions of the SA module.

The ACL is generated by the SPC at compile-time. The compiler uses the interface definition of a software module to identify CDCs. A CDC caller is identified by the SPC by searching for the abstract interface calls within the annotated sources and using the role assignment to map it to an SA. But given that the SPC has not a CFG on function level, it cannot automatically resolve the precise SA. Hence, all SAs that use the software module get access to the CDC. The current compilation model generates a very coarse-grain ACL that needs a manual specification yet.

Due to the fixed program structure of the Meetering app, run-time modifications are not necessary. We place the ACL in the same data space that contains the private variables of the SA module. The data structure of the ACL is given by Listing B.3 in Appendix B.1.

6.2.3 SA stack isolation

Because of the single address space of the MSP430 ISA, most of all OSs use a single stack that is shared between activities. The use of a sole stack simplifies the implementation of the OS, so that on an activity switch the stack can be used unchanged. But a sole stack has a significant impact on the system's security and safety. An isolation of SAs with individual stacks for each activity reduces side effects in a significant manner. But the isolation requires an estimation of the stack size of each SA to provide a suitable portion of memory. The stack size can be estimated by using different techniques, as presented by Togerson [Tog05] or Regehr et al. [RRW05].

The langOS OS library implements the concept of co-routines and does not provide individual SA stacks. Hence, we had to introduce a modified memory layout to hold SA stacks. Since each SA has an isolated data space that is assigned to it exclusively, we can isolate private data needed by the security platform without adding an additional data space. Instead, we extended the data space that holds the private data and structured it in a data, a stack, a canary, and an ACL segment. We placed the ACL at the top of the data space above the stack to avoid an overwriting in case of a stack overflow. Furthermore, we protect the ACL by a canary word to detect stack smashing. The extended memory layout of a langOS application is shown in Figure 6.7. Due to the fact that our Meetering app has no dynamic memory management, a fixed program flow, and no nested interrupts, the segment sizes can be estimated at compile-time.

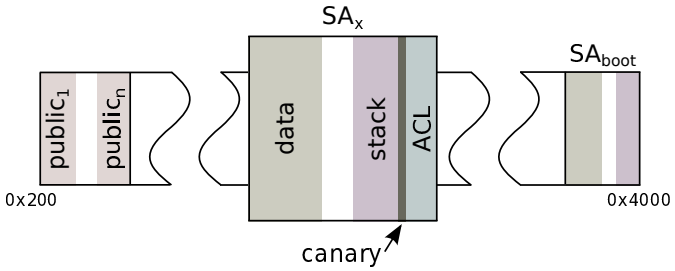

Fig. 6.7: The memory layout of langOS applications provides individual stacks for the SAs. The stacks are placed in a data space combined with private data and the ACL. The ACL is protected additionally by a canary word.

All SAs have the same memory layout except the boot-strap SA. Due to the fact that the boot-strap SA has no incoming transitions, it has no ACL, so that the stack can be placed at the top of the memory. Hence, no modifications on the tool chain are necessary.

6.3 Sealing an embedded controller application

We introduced the SWUR application as a second example of TSS applications in Section 2.5.2. In contrast to the Meetering app the SWUR firmware is not based on any OS library as langOS. Hence, the software was developed by considering the capabilities of the secure platform based on the tinyVLIW8 soft-core processor and its CAM-based MPU. Because of the extensive hardware requirements of a large CAM-based DDT the initial tinyVLIW8 implementation is limited to 32 DDT entries.

6.3.1 Tiny scale embedded controller

To improve the over-all efficiency of the SWUR a subset of the software components were implemented in hardware modules. Figure 6.8 shows the block diagram of the hardware-based SWUR. The system architecture is based on the tinyVLIW8 soft-core processor that implements the TOTP algorithm. The soft-core processor is extended by a hardware-based SHA-1 and a symbol decoder. The symbol decoder samples the wake-up signals from the wake-up receiver and decodes the received wake-up stream. A detailed description of the symbol decoder is out of scope, but can be found in Stecklina et. al. [SKM14].

Beside the high energy efficiency of the SWUR the system's security is improved by the hardware-software co-design as well, because the security critical information is stored in-

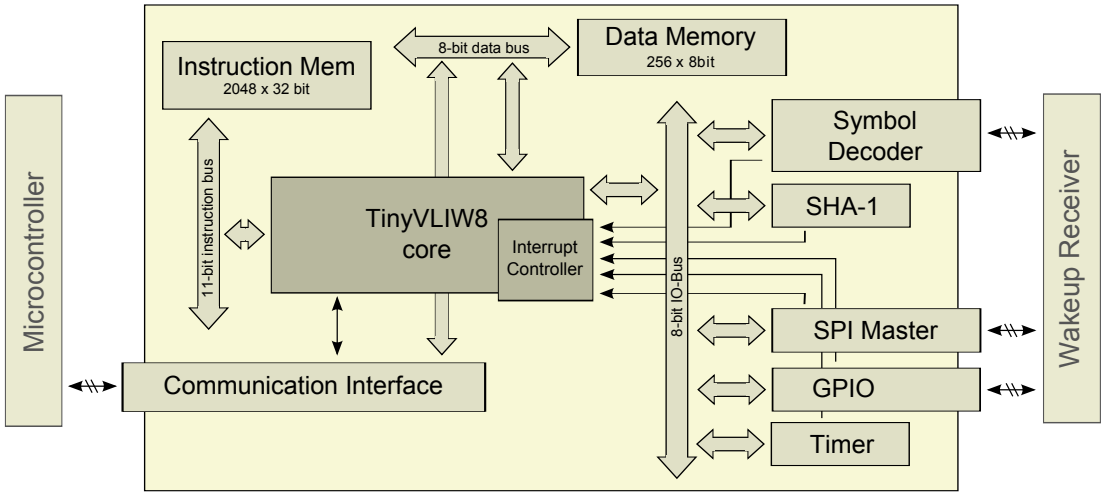

Fig. 6.8: System architecture of the hardware module of a SWUR.

side the hardware module. The MCU's firmware configures the SWUR hardware module at start-up and uses it for the generation of the TOTP value later. Due to the fact that the implementation of the HMAC algorithm is part of the tinyVLIW8 soft-core processor, the secure key and the generated hash value are not accessible by an external MCU. External components have access only to the communication interface that provides a configurable register windows, which maps dedicated sections of the tinyVLIW8 system memory.

6.3.2 SWUR firmware

The SWUR firmware is implemented from scratch without using any OS library. The software architecture was mainly constrained by the MPU characteristics. The call graph of the SWUR firmware is shown in Figure 6.9.

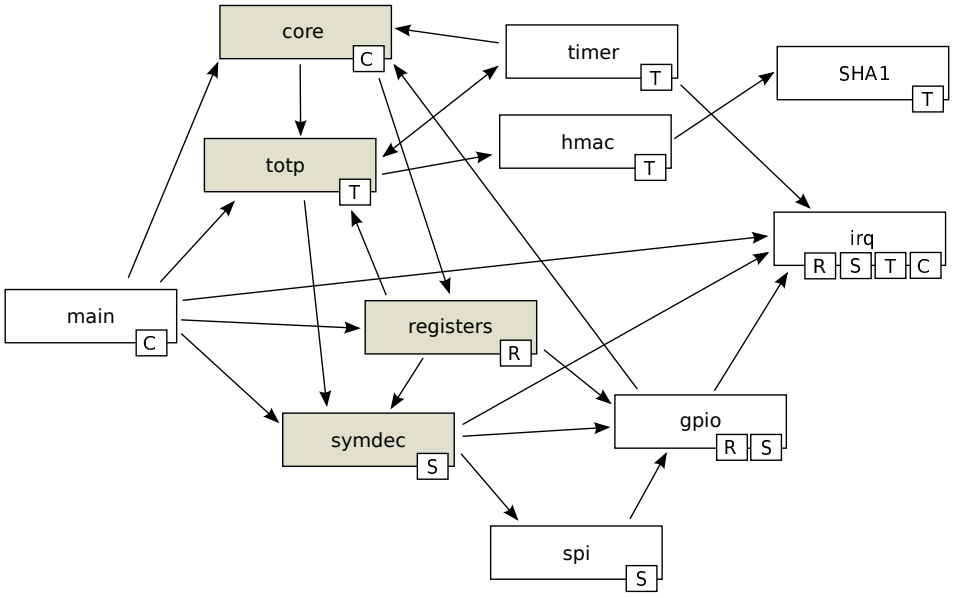

Fig. 6.9: Call graph of the SWUR firmware. The small boxes show the SAs that use the module.

6.3.2.1 SPD of the SWUR firmware

For the SWUR firmware four different SAs were identified. In detail the SAs are defined as follows:

core The firmware initialization and the main loop are implemented by the *core* SA. It needs access to the init functions of all other SAs. A very small scheduler is used to call interrupt bottom-halfs.

totp The *totp* SA implements the TOTP algorithm and encloses timer and HMAC functionality. The Secure Hash Algorithmus 1 (SHA1) is accelerated by a hardware unit, so that the totp SA needs access to the MMIO interface of the SHA1 unit.

reg The configuration interface is isolated by the *reg* SA. It needs physical access to GPIO resources only. Further operations must be called by CDCs.

symdec The symbol decoder driver is implemented in the *symdec* SA. It uses the SPI master for the communication to the wake-up receiver (WUR) IC.

The complete SPB of the SWUR firmware is shown in Listing A.2 in Appendix A.2.

The software modules were mapped to six roles: C, R, T, S, RS, and $RSTC$[6]. Since one role is shared by two SAs and one role is used by all SAs the six roles can be mapped directly to five data spaces. The data space shared by two SAs is owned by the *symdec* SA and granted to the *reg* SA. The data space used by all SAs is also owned by the *core* SA and granted to the *zero* SA, so that all other SAs have access to it.

Table 6.4: The SWUR firmware defines 18 data spaces in case of using a CAM-based DDT.

	Number of data spaces
code segments	5
private variables	4
public variables	1
IOMEM	6
MPU MMIO	1
nucleus gate	1

We have defined a private data space for each SA, which include all global variables of the SA module, the ACL, and the private stack. Due to the fact that the tinyVLIW8 soft-core processor provides only an emulated function call (see Listing 6.2 in the following subsection), the software has full control on the stack handling, which simplifies the implementation of SA stacks in a significant manner.

The firmware uses only one public variable, which is shared between two SAs using a call by reference CDC. We defined an additional data space for this public variable. The data space

[6]We used the first character of the SA names to build the role names.

is owned by the caller and is statically granted to the callee SA. Thus no run-time access to the MPU management registers is necessary. Hence, we can use a sole data space to grant access to the SAID register to all SAs.

We isolated the MMIO resources listed in Table 6.5 by using a dedicated data space for each peripheral resource. Therefore, six different data spaces are required. Although the *totp* SA needs access to the timer and the SHA-1 registers, we have to define two different data spaces, given by the fact that their MMIO areas are not consecutive.

Table 6.5: Memory map of the IO resources of the tinyVLIW8 soft-core processor.

Module	Base address	Size
IRQcntrl	0000 0001 (0x01)	1 byte
SPI	0001 0100 (0x14)	4 byte
SHA1	0001 1100 (0x1c)	4 byte
GPIO	0010 0000 (0x20)	8 byte
Timer	0010 1000 (0x28)	8 byte
MPU	0011 0000 (0x30)	8 byte
SymDec	0100 0000 (0x40)	16 byte

The nucleus gate is implemented within an isolated data space, which is granted to the zero SA to guarantee access for all SAs. We might share the nucleus gate data space with the data space used for the interrupt code segment, so that one data space could be saved.

6.3.2.2 tinyVLIW8 CDC optimizations

The tinyVLIW8 soft-core processor does not feature a function call instruction. The function call must be emulated. Listing 6.2 shows the six VLIW assembler code instructions of a function call emulation. In detail the emulation has to copy the PC onto the program stack, which is addressed by register R7. The current PC is available via the MMIO interface. It is mapped at the addresses `0x06` and `0x07`. Before pushing the current PC onto the stack, it must be incremented by 4 to take the following instructions into account. Afterwards, the processor can jump to the function. The jump is performed directly by using the `jmp` instruction.

In an analysis of the program flow, we have identified 11 CDCs. In case of implementing a generic CDC as proposed in Listing 5.2 in Section 5.2.2, a significant program overhead is caused on the tinyVLIW8 soft-core processor. Furthermore, in case of using the common structure of Listing B.3 for the CDC array, 28 % of the available memory will be consumed. Hence, a more optimized CDC implementation and a more efficient ACL storage is necessary.

Listing 6.2: Function call emulation of the tinyVLIW8 soft-core processor.

```
ldi r4, #0x06 | add r7, #0xff;
ldi r5, #0x07 | add r4,  #0x04;
st r4, @r7    | addi r5, #0x00;
add r7, 0xff;
st r5, @r7    | jmp __func__;
```

We implement CDC optimizations by using an extended compile-time approach. As already mentioned the grant operation is not used by any SA of the SWUR firmware. Therefore, we

can skip the granting of parameters in all CDCs. In addition, the ACL can be implemented directly in the CDC by checking the content of the `SAID` register and the `DSID` register statically. Hence, a generic CDC within the nucleus gate is not necessary on the tinyVLIW8 soft-core processor. So the memory required to store the ACL and the instructions needed to perform a CDC can be reduced significantly.

6.3.3 Configurable compiler suite

The fundamental problem of a specific hardware ISA is its dual use as the persistent representation of software and as the interface by which primitive hardware operations are specified and sequenced. By using a virtual ISA further information can be included and can be preserved for a later use. The information may include typed registers, an explicit control-flow and data-flow. Current build tools use an intermediate language to get an independent program representation. The LLVM project uses a virtual instruction set (VIS) based on a lightweight virtual machine to enforce platform independent program optimizations [LA04].

A similar approach is provided by the CoMet tool. The tool chain is shown in Figure 6.10. It uses a front-end compiler to transform C sources into an intermediate code. In contrast to LLVM, the CoMet tool chain is able to simulate intermediate codes with the integrated simulator on each transformation step [USV+15]. Due to its flexibility and its simulation capability we analyzed the CoMet tool chain regarding its capabilities to generate a security enhanced firmware for the tinyVLIW8 soft-core processor.

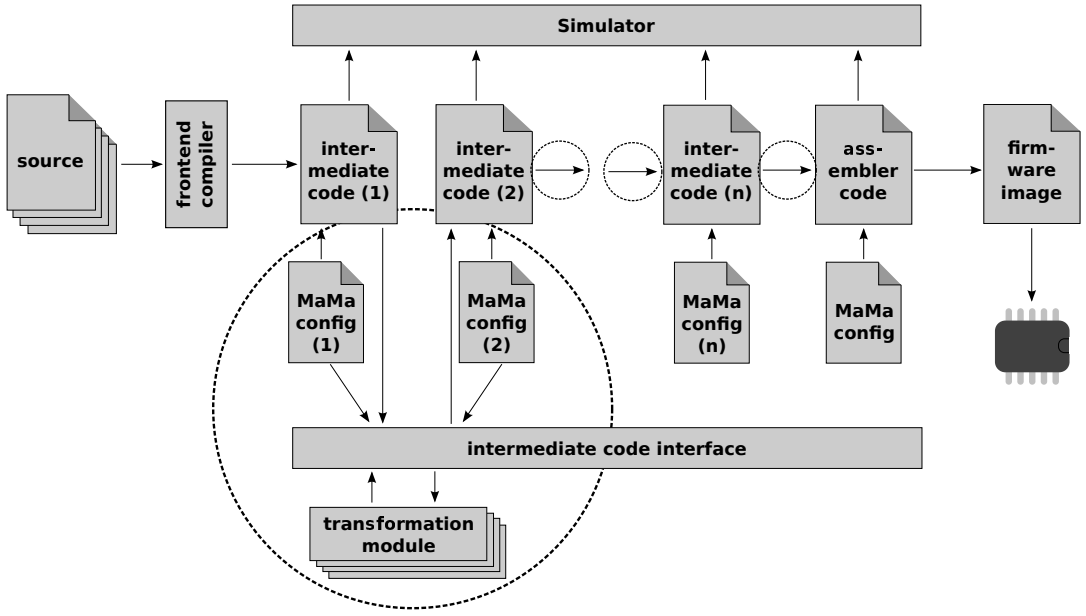

Fig. 6.10: Configurable design process by using the CoMet tool chain [USV+ 15]. The source code is transformed into a final firmware image by using transformation modules. On each transformation, parametrized by a MaMa configuration, simulatable intermediate code are generated.

The intermediate code interface uses the MaMa configuration and the current intermediate code to transform the current intermediate code in an alternative intermediate code. Each transformation module is implemented in a program library that can be added dynamically to the tool chain. We added a set of transformation modules to build a tinyVLIW8 compiler that generates assembly code. The modules are fine-grained to make additional optimizations

feasible. We have implemented modules for generating 16-bit code, mapping 16-bit code to 8-bit, adding complex mathematical operations, register allocation, static memory allocation, and function calls.

In an extension of the transformation modules features of our security enhanced platform can be integrated. Based on the SPB we can generate a memory layout with isolated data spaces, an optimized register allocation to speed-up CDCs, and we can integrate CDCs with an ACL check in function calls based on the demands of the SPD. In contrast to the extension of the langOS compilation model the CoMet tool makes a use of cross-compilation information possible. Additional information can be placed in the intermediate code. The MaMa configuration allows interpreting these information in the next transformation step.

CHAPTER 7

Platform evaluation

We introduced the concept of a platform for an implementation of security enhanced TSSs. The benefit of the presented platform will be measured by its security gain and its cost in performance and physical resources. In this chapter we will give a qualitative, high-level evaluation of the security of our secure platform first. In the following, we will discuss the platform's cost in values of performance and resource utilization.

7.1 Security evaluation

We mentioned in Section 1.1 that the system's security can be verified by a formal proof. Such a proof was done by Klein et al. for the L4 μ-kernel [KAE+14]. But formal proven systems are used very rarely, which is mainly caused by the lack of an intuitive and a practical way of correctly employing it. Abadi et al. [ABEL05a] mentioned that it is difficult to quantify the security benefits of any given technology. The effects of unexploited vulnerabilities cannot be predicted and real-world attacks might thwart any mechanisms by trivial changes to those details.

In Chapter 2 we prepended a description of goals and assumptions to the introduction of our concept of a secure platform for TSSs. We described local and non-local attacks and clarified that local attacks are major critical for TSSs. Moreover, we introduced six technologies to build secure systems. In the following, we give a qualitative security evaluation, in which we will show that our platform provides countermeasures against local attacks and enables the building of secure systems based on the six technologies presented by Hermann Härtig [Här02]. Our security evaluation is divided into three parts:

- an evaluation of platform technologies to counter threats and weaknesses,
- an analysis of the applicability of security techniques to build secure systems, and
- a comparison with state-of-the-art technology presented in Chapter 3.

7.1.1 Platform security evaluation

The implementation of secure software systems must be founded on a secure platform. The most distinguishing feature of our secure platform is providing of a secure isolation of software activities and a trustworthy access control on function level. We presume that our platform does not have any intrinsic weakness, which would give an adversary an unauthorized access to foreign resources. Hence, we will focus our security evaluation on threats and weaknesses that are part of the applications.

First, we evaluated the security of our platform by investigating measures to counter local attacks as described in Section 2.1. We introduced two common ways to arrange malicious code within the victim's address space. The code can either be injected by using an unprotected buffer or can be constructed out off existing code by manipulating the program flow, e.g. by using gadgets as presented by Francillon et al. [FC08]. Hence, an adversary needs either write access to a memory section to inject malicious program code or needs execution permissions to a sufficient amount of program code to build a malicious sequence. In the following, we will expose three basic features of our secure platform, which help to reduce the size of the attack vector of this type of attacks.

7.1.1.1 Augmented memory sections

A TSS built on a common MCU with a von-Neumann architecture uses a single address space, in which all data sections are handled in the same manner, so that the injection of additional program code or the reuse of existing code is not forbidden by any mechanism. A Harvard-architecture provides a separation of the data and the program section, but Francillon et al. [FC08] have shown that any piece of code can be injected under convenient circumstances as well.

The here presented secure platform for TSSs provides an augmentation of memory sections in such a manner that attributes such as readable, writable, or executable and any combination of these attributes can be assigned to each memory section. This feature ensures that data sections are not executable and code sections are not writable. Hence, an execution of program code out of a writable data section or any modification of a code section can be prohibited effectively. The implementation of permission rights is a fundamental capability of systems to enforce security and safety.

Nevertheless, the execution of gadgets might still be possible. But the isolation of SAs has a direct impact on the size of the computing base and on the size of the available stack. Small program sections require a small program stack, which makes the installation of gadget chains much more difficult.

7.1.1.2 Reduced computing base

The separation of SAs opens up the possibility to reduce the amount of code that has access to critical resources. Especially in the context of CPSs, a separation of the communication, the computation, and the control component is a mandatory measure to build secure and dependable systems. Furthermore, Goodspeed et al. [GF09] analyzed the probability of finding a gadget for a ROP attack and figured out that especially the number of functions has a direct impact on the probability of the availability of suitable gadgets. Since ROP attacks are always tailor-made for an MCU and for an application, a threshold for a secure and insecure system cannot be given. But we can presume that the isolation of SAs reduces the size of the computing base within a protection domain, so that the attacker's chances of success are reduced likewise.

We analyzed our example applications to give a qualitative evaluation to see the impact of the isolation of SAs on the number of modules, functions, lines of code (LoC), and code size. The measured values for the Meetering app are summarized in Table 7.1.

Table 7.1: Modules, functions, LoC, and code size of the SAs of the Meetering app.

SA	modules	functions	LoC	code size (bytes)
RADIO	10	160	3,452	16,380
BOOTSTRAP	17	161	3,272	14,626
MAIN	8	94	1,614	9,144
STORAGE	8	85	1,963	10,790
CAPCTRL	2	31	500	3,318
NETWORK	8	113	1,975	10,084
all	26	272	6,030	25,896

The values of Table 7.1 show that the CAPCTRL component, which is the most valuable component, has the smallest computing base of all components. It includes only 13 % of the code size and 8 % of the LoC of the Meetering app. The probability of an error within this component is quite low. But it is also shown that the most vulnerable components, the RADIO (57 % LoC) and the NETWORK (33 % LoC) SA, have still a large computing base. The large computing base is caused by the structure of langOS. As already mentioned in Section 6.1.1, langOS was developed with a focus on resource efficiency. Hence, it includes modules that concentrate functionality and interact with a large number of software modules, so that these modules are used by a large number of SAs and the amount of shared code is very large. Furthermore, the structure of langOS prevents a suitable separation of the module initialization and the run-time functionality and makes a specific isolation of SAs, with security benefits, difficult.

However, we cannot achieve a small computing base for the most vulnerable components. But we can ensure that most valuable component is isolated from it. All the security threats of the Meetering app, presented in Section 2.5.1.2, are sourced by the CAPCTRL component, so that a protection of this component is key. Hence, we are convinced that our platform reduces the remaining risk for the Meetering app, induced via its public interface, in a significant manner.

The measured values of the SWUR firmware are given in Table 7.2. We measured the number of instructions instead of the code size. The pure code size may include VLIW instructions, which are not executed, so that a measurement of code size is less meaningful.

Table 7.2: Modules, functions, LoC, and instructions of the SAs of the SWUR firmware.

SA	modules	functions	LoC	instructions
CORE	3	6	104	109
TOTP	5	19	241	541
REG	3	10	133	159
SYMDEC	4	16	212	285
all	11	39	465	899

The SWUR firmware allows a better isolation of software activities. It includes only twelve shared functions with 200 shared instructions. Only two software modules are shared between SAs. Hence, the computing base of the modules is reduced equal to the number of SAs. The only exception is the TOTP SA, it includes 60 % of the instructions and nearly half

of all software modules. It implements the TOTP algorithm, which does not allow a more fine-grained separation of software modules. But we can see that the computing base of the most vulnerable component, the REG SA, is reduced significantly. It includes 26 % of all functions and only 18 % of all instructions.

Both example applications have shown that the computing base of SAs can be reduced significantly. Obviously the software structure has a large impact on the enforcement of a small computing base. In case of a large amount of shared functionality or tightly coupled software modules, the computing base may still be large. Hence, the enforcement of a suitable isolation to reduce the computing base requires a well-defined software structure with the security platform in mind.

7.1.1.3 Privilege separation

The privilege escalation uses a weakness in hardware or software or a configuration error to gain additional access to resources that are normally not assigned to a software entity. Since security enhanced TSSs do not provide vertical privilege levels, a horizontal privilege escalation becomes even more critical. A horizontal privilege escalation will be used by an adversary to get access to resources that are assigned to another SA.

In our security platform for TSSs we can identify two possible methods for a horizontal privilege escalation:

Unauthorized function calls An adversary can invoke a function of another SA to get access to their resources. This is possible if an SA is able to bypass the access control or to call a non-public function of a victim SA.

Imposing functionality An adversary can impose program code into the protection domain of a foreign SA to perform the biddings of itself.

Unauthorized function call

The protection of a function call is based on a secure management of the ACL. It must be ensured that an adversary cannot manipulate the ACL to its own benefit. Hence, it is mandatory that the SAID of a caller is provided by a trustworthy instance.

We have integrated an access control check in a CDC to ensure that a public function can be used by a certain SA only. The check is based on an ACL generated at compile-time by the SPC. Since the ACL is stored within a private data space of the callee's protection domain, a malicious SA has no access to the ACL. Furthermore, since the ACL is built by the SPC, at compile-time a modification of the ACL at run-time is not necessary or foreseen. A secure protection of the ACL of an SA requires a secure isolation of data spaces. Hence, the proposed access control is strongly coupled with a secure and trustworthy implementation of the underlying MPU.

A domain switch is executed by committing the new SAID to the MPU. The domain switch is performed immediately, so that it is demanded that the subsequent instruction can be executed by the new SA. The SAIDs are stored on the SAID stack, which is part of the

memory protection nucleus. Figure 7.1 shows the separated access on the stack elements. The MPU has read access to the callee element, which is the element last recently pushed on the stack. The software has only read access to the previous element, which contains the callee. The element is used to check access on function level. A write access is performed on the element above the callee, which moves the rotation window one element forward. The caller does not have any interface to manipulate this mechanism. Hence, the SAID of a caller cannot be faked.

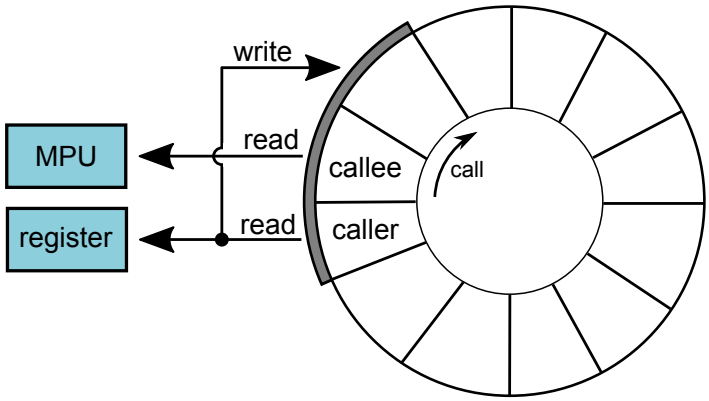

Fig. 7.1: Register and MPU access on the SAID stack. The access on the rotating window is separated depending on the access source (MPU/register) and access type (read/write).

Imposing functionality

The fixed program flow of the nucleus gate leads us to the most critical method of privilege escalation within the proposed security platform. A malicious SA can try to prepare a code section that is executed by a victim SA to perform operations within a foreign protection domain. The attack is illustrated in Figure 7.2.

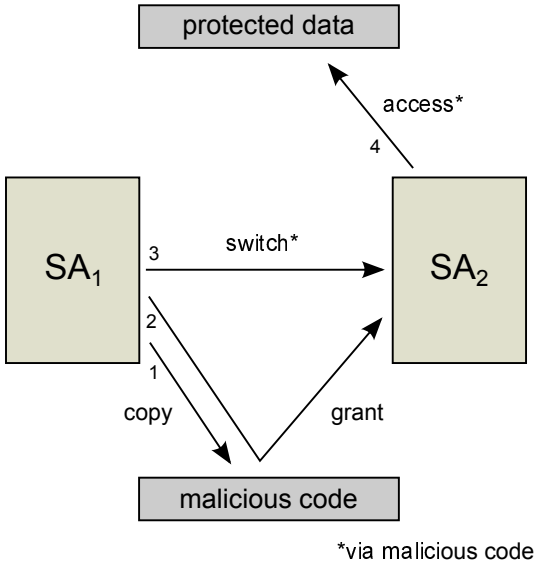

Fig. 7.2: Imposing a data space on a foreign SA to perform biddings of an attacker.

The attack, illustrated in Figure 7.2, consists of the following steps:

(1) A malicious SA creates a new data space and copies the malicious program code into it.
(2) The data space is granted to the victim's SA.
(3) The malicious SA switches to the victim SA by using the imposed data space[1].
(4) The malicious code accesses the protected data space to get information or perform restricted operations.

The problem is caused by the possibility of an unrestricted delegation of data spaces to any SA. It can be solved by two alternative strategies: a trustworthy control instance or a two-way handshake. The two-way handshake is implemented by the L4 μ-kernel. A page can be delegated to a foreign activity only if it expects a delegation. A similar implementation could be integrated in our approach with small modifications on the delegation concept. But the approach would require an additional domain switch. Instead, we have proposed a trustworthy instance that is used in case a non-trustworthy instance needs the capability to delegate a data space. In Section 5.2.2 we introduced the trampoline CDC, which performs the role of such a trustworthy instance.

The imposing of a data space can be avoided by forbidding non-trustworthy SAs any access to the `ADDR` and the `DATA` register of the MPU. Such an SA cannot delegate any data space, it has to use the trampoline SA instead. However, the approach requires an additional domain switch to the trampoline SA, it can be used selectively and data space delegation can be kept as simple as it is.

7.1.2 Implementation of security techniques

In Section 2.3 we introduced six technologies to build secure systems and in Section 4 we proposed four basic principles of our secure platform for TSSs. In the following, we will illustrate in detail the enforcement of each technology by using our basic principles of our secure platform. Table 7.3 gives an overview about cross-points between the security technologies and the basic principles.

Table 7.3: Cross-matrix of the security techniques small interfaces (SC), access control (AC), tunneling (T), secure boot (SB), resource control (RC), and virtual machines (VM) and the four basic principles of the proposed secure platform for TSSs.

	SC	AC	T	SB	RC	VM
Data spaces			x		x	x
Flow integrity		x		x		
Trustworthy instance	x		x		x	
Access control		x				

7.1.2.1 Small interfaces

A proven key principle of a secure design is to prohibit access to all resources by default and allowing access only through well-defined entry points, i.e. interfaces [Ven05]. Hence, the system's security is mainly affected by the design of interfaces. The interfaces of our

[1] Due to the fact that a common SA switch, provided by the nucleus gate, is always combined with an ACL check, the imposed code section must be used to switch to the victim's SA.

proposed secure platform are determined by the interface of the security nucleus and the number of public functions of SAs.

Interface of the security nucleus

The interface of the security nucleus was inspired by the L4 μ-kernel, which provides only few system calls and requires an implementation of complex operations in user space. Hence, the interface of our memory protection nucleus is limited to the data space management (grant and map) and the domain switch. Extended functionality must be implemented by software components in isolated SAs.

The functionality of the software-based nucleus gate depends on the system architecture and on the SPD definition. We implemented a complex nucleus gate for langOS with an extended interface. The interface provides only seven functions, which is similar to the small interface of secure μ-kernel systems, such as L4. In addition, we have shown in case of our tinyVLIW8 processor that the functionality of the nucleus gate can be fully integrated in the application as well. The proposed security platform makes no restrictions on this and allows an implementation of a kernel-free system tailor-made for TSSs with limited resources and real-time requirements.

Application's interfaces

Depending on the system's implementation, the interface provided by SAs can be quite small. But the enforcement of small interfaces requires an appropriate software design. We suggested a stream-like event processing in which an SA performs a specific task and transfers control to the next SA. Considering our secure platform and the constrains of TSSs, an application with very small interfaces can be designed. It must provide a very small interface for data reception only. Further operations may be performed within the protection domain of the SA.

As an example, we will have a look on langOS, which was implemented without considering our secure platform. It was focused on a small code size. Therefore, the call graph of langOS applications are not organized stream-like and need further improvements. A better enforcement of small interfaces will be a trade-off between small code size and public (shared) functions. We identified the network stack as a component with a large amount for shared functions. In a secure application the interface of an network stack layer must provide only two functions: `receive` and `send_done`. In addition, the two functions must include a pointer to the packet buffer. In such a system, a packet buffer might be owned by one SA and access for all others layers can be limited on an appropriate access. The interface will be more secure, so that local vulnerabilities of a component can be restricted to this one.

7.1.2.2 Access-control based on contracts

The enforcement of access control on resources is a core component of our proposed platform. We have presented an MPU that controls access on memory resources and an ACL-based access control on function level.

Since cross domain communication is mandatory in distributed systems, we introduced two alternative technologies: either data spaces can be delegated to a foreign SA or an SA can

call a function of a foreign SA. Both communication channels are controlled by an adapted RBAC scheme that makes a fine-grained access control within TSS applications possible by using an application-specific SPD. The SPD describes the application-specific contract among the SAs. It is generated at compile-time and is based on an analysis of the program flow that identifies the interactions between SAs. Furthermore, in case of using langOS run-time bindings can be avoided, the compile-time analysis covers the majority of the possible run-time variants.

Due to the fixed definition of the ACL an SA can access only those resources that are assigned to it at compile-time. But it is mandatory to store the ACL within the protection domain of an SA to prevent any foreign access. Otherwise it cannot be guaranteed that it is not manipulated by a malicious SA. Hence, a private data section must be assigned to an SA that uses an ACL to control access on function level.

7.1.2.3 Tunneling

The separation of SAs causes a separation of functionality as well. Hence, in a system with isolated SAs a single SA is often forced to use an additional SA to perform a demanded task. But a tight coupling of SAs to perform a joint task increases the probability of a vulnerability. Therefore, a technology is necessary, which allows the use of additional functionality with a strict and secure isolation.

The tunneling technology enables an SA to pass data through a foreign SA with the guarantee that the data cannot be manipulated. The technology is in common use for enforcing integrity in communication channels, which are using an insecure communication medium. We can provide a limited view on data spaces, so that a foreign SA has limited access to it. Tunneling can be implemented by the use of shared data spaces. Due to the fact that different capabilities can be assigned to different SAs for a single data space an asynchronous sharing becomes possible. Hence, an SA can delegate a set of capabilities tailor-made for the given task.

In TSS applications the tunneling technology becomes useful if an SA uses a complex module to process trustworthy information. Since the complex module may be vulnerable, its SA must be seen as an untrustworthy component, which does not have access to the DDT management. A trustworthy SA can delegate a data space to the complex SA and give only appropriate access to critical data sections. In the following the complex SA can manipulate only a defined subset of the data space and a manipulation of critical data can be prevented. An example of such a complex SA might be a network protocol layer. Its access on the packet data can be limited to the header information, so that the integrity of the payload information can be guaranteed.

7.1.2.4 Secure boot

The secure boot technology implements a boot chain of software components, in which each component verifies the integrity of all components, which are invoked by itself. The secure boot system is based on a trustworthy root anchor, which is known to the initial process only. The trustworthy root anchor must be provided by hardware. In common computer systems the root anchor is often stored in the trusted platform module (TPM).

The enforcement of security in TSS is mainly based on the definition of the SPD. The application-specific SPB as part of the SPD includes a tailor-made definition of SAs, roles, and role transitions. Especially the definition of role transitions makes an enforcement of a reliable chain of software modules possible. However, the presented approach does not include a trustworthy root anchor, which is mandatory for a secure boot. Instead it provides mandatory technologies to build a secure boot chain. The trustworthy root anchor can be backfitted, as done in common computer systems.

El Defrawy et al. [ETFP12] propose SMART, a simple and efficient hardware-software primitive to establish a dynamic root of trust in an embedded processor. The SMART approach is based on two hardware modifications: *key access control*, to enforce key protection, and *ROM execution control*, to restrict execution of ROM code. Since both features are provided by our platform, we are convinced that an implementation of a boot-scheme as provided by SMART can be implemented on our security platform as well.

7.1.2.5 Effective resource control

Driven by the work in real-time systems, effective resource control against denial-of-service attacks has brought significant progress in two areas: better control over the allocation of resources and early demultiplexing [Här02]. Whereas early demultiplexing is focused on protocol stacks and newer implementations even make use of small dedicated hardware units, control over the allocation of system resources has an impact on the whole system and can be enforced by providing data spaces and a trustworthy instance.

The allocation of system resources is integrated in our security nucleus. An SA needs access to the MPU to allocate a new data space or has to use functions of the nucleus gate. In both cases the access can be restricted to individual SAs by defining the definition rights to the MPU registers, so that exhaustion of system resources by malicious SAs can be prevented. Hence, an efficient resource control is given intrinsically by our platform.

7.1.2.6 Virtual machines

Virtual machines are an effective instrument to provide a secure isolation. Security-focused OSs aim to provide security through isolation based on virtualization, e.g. *Qubes OS* [RR10]. Sensor node OSs provide virtualization to implement safe systems or to make a secure update of software components possible.

An efficient virtualization can be provided only if a guest is running directly on the host's CPU. But a native execution requires an installation of different privilege levels. Since common MCUs, e.g. the MSP430, use a single address space, a VMM cannot guarantee that it retrieves control on the CPU after enabling a guest. A secure isolation of system resources as provided by our platform makes a native execution possible. A guest can get limited access, so that it cannot escape from its virtual environment. Furthermore, the VMM can use system resources to retrieve control periodically by retaining an exclusive access on system resources such as timers or interrupts.

Even though our platform provides the technology to implement a native virtualization, a virtualization of OSs will still be expensive on TSSs. A virtualization of software components or small applications as provided by safe languages is a promising approach to provide an

enhanced security on dynamic TSSs. But an evaluation of such an approach to provide a more flexible security platform is out of scope of this thesis and might be investigated in further activities.

7.1.3 Comparison with state-of-the-art of technology

This thesis presented a tailor-made MPU for TSSs. In Chapter 3, we discussed various concepts for a secure memory protection in general-purpose systems and in embedded systems. Whereas historical systems define fundamental mechanisms used in most of the following systems, our approach makes use of mechanisms that are common in general-purpose systems. A discussion of the security mechanisms will be done in comparison with the mechanisms designed for embedded systems.

7.1.3.1 Hardware-based memory protection schemes in TSSs

Powerful processor cores, as ARM11, ARC core, LEON-2, or Nios II, provide a memory protection comparable to common computer systems. Since these systems implement a fully featured MMU, with extensive resource utilization on FPGA devices, they are out of scope of our evaluation. Instead, we will compare our MPU with approaches designed for MCUs of deeply embedded systems.

Infineon embedded processor The Infineon TriCore MCU was used by Lohmann et al. to implement a configurable memory protection by aspects [LSH+07]. The KESO approach makes use of a peculiarity of the TriCore MCU. The MCU does not disable memory protection when the processor runs in supervisor mode. Thereby, the approach exploits this peculiarity by running even the application code in supervisor mode, so that only the MPU must be reconfigured to perform a domain switch. Hence, the presented approach is very similar to the fast domain switching of our platform. An MPU protection can be implemented by banning untrustworthy applications in user mode. These applications must trap into the system kernel to perform privileged operations. Although such a mixed mode is not proposed by Lohmann et al., we are convinced that the approach can be modified suchlike very easily. But the Infineon TriCore features a small set of segment registers only. Thus, a fine-grained memory protection cannot be enforced.

MSP430 FR57xx family The MSP430 FR57xx family is limited in its number of available segments. It implements only three segments, which makes an enforcement of a fine-grained isolation of SAs impossible. In constrast to the Infineon TriCore the MCU provides a protection of the MPU registers, so that an isolation based on software might be possible.

Lopriore MPU The approach presented by Lopriore is concerned with safety rather than security. A deliberately harmful subroutine can extent its own local context up to the limit of the entire global context and will get access to resources that it does not owns [Lop14]. The approach does not include a mechanism to protect the MPU against

a harmful use as well. Furthermore, the memory space is logically partitioned into 2^n fixed size blocks, so that a fine-grained isolation can be implemented with a significant overhead only.

Mondriaan memory protection (MMP) The MMP is a fine-grained protection scheme with flexible memory segments. The segments can be exported to provide a protected view to other protection domains. It allows also an individual permission control. The feature set of the MMP is similar to the MPU presented by this work. We are convinced that the system allows the implementation of the six security techniques as well. But the MMP was designed for Mondrix, a modified version of Linux, and its implementation was integrated in the Bochs emulator only. To the best of our knowledge a real hardware implementation is not available yet.

Micro memory protection unit (UMPU) The UMPU presented by Kumar et al. was implemented on an AVR ATmega 8-bit MCU and provides a segmentation of the system's memory [RSC+07]. Furthermore, protection domains export functions to enable cross domain calls. The approach provides three of our basic principles completely or partially. But the approach is focused on safety and a sufficient definition of access control is not available. Hence, an SA cannot delegate a limited view on its data spaces to foreign SAs, so that an implementation of secure systems is not possible.

Sancus The security architecture of Sancus is focused on networked embedded systems and provides an extended protection of data sections on an MSP430-compatible soft-core processor. The isolation of software modules is implemented by restricting the access to the protected data of a module so that it is accessible only while the program counter points to an address within the text section of the same module. But an implementation of shared memory is not possible. Furthermore, the system requires an underlying operating system, which may be potentially insecure.

We can conclude our comparison with the finding that all these approaches feature safety or security with different goals. The hardware-based approaches are more restricted in the number of protection domains or provide a more coarse-grained isolation than our approach. Even though we identified features such as fast domain switching, which are provided by our platform as well, whose implementation lack the applicability for security. Hence, an approach with an comparable set of security features could not be identified.

7.1.3.2 Software-based memory protection scheme

Aiken et al. have shown that a language- and compiler-based memory protection scheme may be more efficient than hardware-based approaches [AFH+06]. We have introduced the tiny hypervisor to provide memory protection on commodity components. Actually, our tiny hypervisor can be considered as a generalized form of SFI. However there are three major differences:

(1) no binary rewriting is used, the binary instrumentation is moved to the transformation phase instead,
(2) our platform allows multiple data spaces assigned to a protection domain, and
(3) our use of dedicated registers is different.

Similar differences exist to the Harbor [RKS07] and the XFI [EAV+06] approaches. But these sandboxing technologies do not enforce security. All these approaches are focused on safety. A write access to foreign data spaces is blocked, but a read access or an execution of injected program code is always possible. Hence, these approaches and all safe language approaches, are not comparable for similar reasons with the tiny hypervisor presented in this thesis.

As motivated and discussed by Rutkowska [Rut08] a software-based security can be provided effectively by a virtualization layer. We introduced Maté [LC02] and SwissQM [MAK07], which feature a VMM on TSSs by implementing instruction emulation. But both approaches provide a very limited ISA to their guests and any guest access to system resources is not allowed. Although these approaches provide a secure isolation, their functional set is very limited, so that they do not allow a simple adaptation of existing programs and a virtualization of a complex SA.

A Java-capable VM can be used to implement complex tasks. Approaches, which provide a JVM for TSSs are presented by TakaTuka [ASE+08], Darjeeling [BLC09], as well as KESO [SSE+13]. But all these systems use an offline compiler and provide a limited VMM that does not feature a strict isolation of guest and host systems at run-time. For performance issues significant parts of the software were compiled to the native instruction set and executed directly on the host processor without restrictions. Security aspects are not focused by Java-capable approaches. Instead they provide safety and portability for TSSs by using the Java programming language.

We can summarize that all the presented software-based approaches either do not provide an effective security or feature a very limited instruction set. The tiny hypervisor here presented overcomes these limitations. It provides an effective security scheme and makes an adaptation of existing, complex programs possible.

7.1.4 Summary

Our qualitative security evaluation started with a presentation of countermeasures to prevent local attacks on TSSs. The measures include an augmentation of memory sections, so that attributes such as readable, writable, or executable and any combination of these attributes can be assigned to system resources. Furthermore, we evaluated the remaining computing base of software components on two real applications. We have shown that the most valuable components can be isolated and their remaining computing base became very small. Finally, we evaluated the horizontal separation of privileges. We illustrated the importance of an access control check on function level. Even though our evaluation did not give a formal proof, we have shown that our platform features a set of security mechanisms to be able to counter local attacks on TSS.

We have shown that it is highly important to have a security platform in mind when designing a secure application. Therefore, we analyzed the applicability of six technologies to build secure systems and we explained how these technologies can be enforced on TSSs by using the four basic principles of our secure platform. Even though our example applications did not make use of these technologies we sketched ideas to improve the application's security by using these technologies in the future. In addition, we have shown that our platform enables the implementation of alternative already proven security approaches without any additional

extensions. Hence, we are convinced that our platform enables an implementation of secure software applications on TSSs.

We completed our security evaluation with a lineup with state-of-the-art of technology. We have shown that most of the available approaches are focused on safety and lack trustworthy mechanisms to build secure systems. Other approaches restrict the feature set of applications in such a manner that complex software systems cannot be implemented. Thus to the best of our knowledge there is no comparable approach with our set of security features.

7.2 Platform cost evaluation

After our security evaluation, we will state how expensive an integration of our platform in real TSSs is. Our cost evaluation will be split into a design size evaluation and a performance evaluation. We will focus our quantitative evaluation on the hardware costs induced on the tinyVLIW8 soft-core processor. The software costs as well as performance drawbacks are evaluated on the MSP430 processor. We will conclude the section with a comparison of the platform cost with state-of-the-art of technology.

An evaluation of the system's power consumption is out of the scope of this thesis. But due to power saving methods cannot be used, such as duty cycling, the power consumption is strongly related to the resource utilization. Hence, we presume that qualitative results of a design size evaluation can be passed to a power consumption evaluation. Especially, the DDT entry look-up engine is active on each instruction, which makes a duty cycling impossible. Hence, the size of the look-up engine will be directly passed to the expected costs regarding the power consumption.

7.2.1 Design size evaluation

The design size of our security platform is mainly driven by the size of the security nucleus (SN). We divided the SN in a memory protection nucleus and a nucleus gate. The memory protection nucleus was implemented in hardware as well as in software. Our design size evaluation will start with an evaluation of the hardware-based memory protection nucleus. In a second part will analyze the size of the nucleus gate. We focus our analysis of the nucleus gate on the MSP430-based langOS OS library. Due to its loose linkage to the OS, the results are likewise applicable for other TSSs OSs.

7.2.1.1 Design size of the hardware-based MPU

Our hardware-based MPU is completely written in the very high speed integrated circuit hardware description language (VHDL) and integrated in the tinyVLIW8 soft-core processor. We used this processor because it was designed by us i.e. extremely well-known and therefore easy to adapt. Furthermore, it is written completely in VHDL. The openMSP430 as an alternative soft-core processor is written in Verilog, so that a mixed design of Verilog and VHDL

is necessary when integrating our MPU[2]. The IHP430X is written in VHDL, but its memory access is partially asynchronous, which complicates the integration of the MPU significantly.

In all of our design size evaluations we set the overhead caused by the MPU in relation to the resource utilization of our tinyVLIW8 processor. Table 7.4 shows the resource utilization by entity of the tinyVLIW8 soft-core processor including the entities used by the SWUR design. The processor was optimized in its design size. Including standard peripherals as SPI, GPIO, Timer, and Debug Interface processor design uses only 1,200 look-up tables (LUTs) and 489 registers. The complete processor design of the SWUR uses 3,643 LUTs and 1,521 registers. We have shown in Section 2.4.2 that common soft-core processors have an at least twice larger resource utilization.

Table 7.4: Resource utilization by entity of the tinyVLIW8 soft-core processor as used within the SWUR design for a Cyclone II FPGA.

Component	LUTs	%	Registers	%
SHA-1	2,166	58.4	847	59.2
tinyVLIW8	**876**	**20.5**	**258**	**15.8**
SymDec	260	7.0	174	12.2
SPI	103	2.6	51	2.7
Timer	92	3.0	64	4.5
DbgInf	68	1.8	67	3.9
GPIO	61	0.1	49	1.7

The MPU includes the register interface, the DDT, the DDT look-up engine, and the SAID stack. Since the resource utilization depends significantly on the number of elements of the DDT and the SAID stack, we will evaluate these components separately.

All measurements of the design size were done with Quartus II 32-bit Version 11.1 Build 258 Web Edition and all designs were synthesized for an Altera Cyclone II EP2C20F484C7 FPGA.

MPU core

The MPU core is the MPU top level entity and includes registers to buffer the DDT commands and the interfaces to DDT, the SAID stack, and the system design. Furthermore, the DSID register is part of the MPU. Table 7.5 shows the size of the MPU in relation to the tinyVLIW8 process.

The complete implementation of the MPU increases the size of the soft-core processor by 228 % in the number of LUTs and by 430 % in the number of registers. The DDT look-up engine includes the combinatorial logic elements of the DDT matching unit and the registers to store the DDT entries. Therefore, it is the largest MPU component and its design has a significant impact on the design size. The MPU core increases the number of LUTs by 47 % . The large size is caused by the complex DDT management operations. The MPU core is the

[2]The ModelSim-Altera simulation software does not support a mixed language design, so that for a simulation of the secured openMSP430 and the MPU a commercial license is required. Due to such a license was not available at any time during the work on this thesis a mixed language design was abandoned.

Table 7.5: Resource utilization by entity of the MPU in relation to the tinyVLIW8 soft-core processor.

Component	LUTs	%	Registers	%
MPU (all)	2,001	228.4	1,108	429.5
MPU (core)	410	46.8	81	31.4
Delay elements	68	7.8	0	0.0
CAM-DDT (32 elements)	1,491	170.2	992	384.5
SAID stack (8 elements)	32	3.7	35	14.6
tinyVLIW8	876	100.0	258	100.0

only trustworthy component that controls the DDT access and has to provide an extended functional set of control operations.

In addition, the current design has a large amount of delay elements to delay the clock signal and the look-up enable signal until result signals of the CAM-based look-up are stable. The number of elements might be reduced by adding an additional clock cycle. But an additional clock cycle would have an impact on the system design, which has to be adapted accordingly.

CAM-based DDT entry lookup

Since the CAM-based DDT has the largest impact on the resource utilization, we will analyze it more in detail and in dependency of its number of elements. The CAM-based implementation of the DDT look-up engine is based on logic elements. Table 7.6 shows the resource utilization of the DDT including the look-up engine. The CAM-based DDT look-up engine uses exclusively flip-flops to instantiate the DDT memory. Therefore, the number of 1-bit registers (flip-flops) correlates with the size of the DDT. When using 32 DDT entries the overall resource utilization is 270 % in comparison to the utilization of the tinyVLIW8 soft-core processor. We chose a configuration with 32 entries as with a smaller number of entries our small example application of the SWUR could not be implemented.

Table 7.6: DDT resource utilization and overhead on the tinyVLIW8 design, when using different numbers of DDT entries in a CAM-based DDT look-up engine.

# DDT	LUTs	%	Flip-flops	%
16	717	81.9	496	192.3
32	1,491	170.2	992	384.5
64	3,001	342.6	1,984	769.0

The advantages of the CAM-based look-up engine are its simple integration and the negligible performance drawback. A parallel DDT entry look-up makes a cache as well as a stall of the processor core unnecessary. Especially the lack of a CPU stall simplifies the MPU integration significantly and reduces the resource utilization. Furthermore, the approach performs a DDT entry look-up in a single clock cycle so that the CPU does not have any performance drawback.

Guccione et al. presented a reconfigurable CAM implementation that takes the technology of the LUT elements of an FPGA into account [GLD00]. The authors have shown that the resulting design will be significant smaller and faster than an elementary logic implementa-

tion. Therefore, we assume that the logic overhead caused by the parallel DDT look-up can be reduced significantly by following the approach of Guccione et al. [GLD00]. The implementation of such a look-up engine is out of the scope of this thesis and has to be focused in further implementations.

DLB-based DDT entry lookup

The DLB-based DDT look-up engine uses FPGA block RAMs to instantiate the DDT memory. The module includes two DLBs, a combinatorial SAID match, and a memory controller to access the FPGA block RAM. As shown in Table 7.7, the number of LUTs and flip-flops is mostly independent of the DDT size. Only the number of memory bits grows with the number of DDT elements.

Table 7.7: DDT resource utilization and overhead on the tinyVLIW8 design, when using a DLB-based DDT look-up engine and 80 bit DDT entries.

# DDT	LUTs	%	Flip-flops	%	Memory bits
32	142	16.2	56	21.7	2,560
64	143	16.3	56	21.7	5,120
128	144	16.4	56	21.7	10,240

In comparison to the CAM-based DDT look-up engine the DLB engine uses only a small number of LUTs. But the DLB engine does not support overlapping regions and requires more memory bits to store the extended capability fields in each DDT entry. In case that we spend 80 bits for a DDT entry, we can include an ACL with two elements and a capability list with four elements. In that case the DDT consumes the memory bits listed in Table 7.7.

Furthermore, the DLB look-up engine requires a processor stall to perform a DDT entry look-up in case of a DLB miss. We have implemented the DDT look-up as a simple sequential search. Hence, the number of clock cycles corresponds to the number of elements. In an average case we get a look-up time of $\frac{(n+1)}{2}$ clock cycles where n is the number of DDT entries.

SAID stack

The SAID stack is implemented as a rotating buffer with a configurable size. Depending on the processor architecture a small number of SAs may be defined. In our introductory example configuration we use eight stack elements only. We synthesized also stacks with a size of 16 and 32 elements. The results are summarized in Table 7.8.

Table 7.8: The resource utilization of the SAID stack in relation to the tinyVLIW8 design, when using flip-flops to store the stack elements.

Elements	LUTs	%	Flip-flops	%
8	32	3.7	35	13.6
16	74	8.5	68	26.4
32	130	14.8	133	51.6

We need at least four flip-flops to store a single SAID. Hence, the size of the SAID stack grows by a multiple of four. In addition, only stack sizes with an order of two are suitable for

an efficient hardware implementation. Since a stack with 32 elements increases the number of flip-flops by nearly 52 % a stack with 16 elements would be a good trade-off between design size and program flexibility. The overhead of LUTs in case of using 16 elements is less than 10 per cent and is acceptable considering the very small size of the processor core.

We did not yet implement an SAID stack with an external memory. Since a parallel access to the current and the previous SAID is needed, the unit must still include registers. Furthermore, the memory controller will need delay elements. Hence, the design size with an external memory might not be significantly smaller.

7.2.1.2 Memory footprint of the nucleus gate

The memory overhead of the nucleus gate is split in a constant overhead and an SPD-specific overhead. In addition, we can differentiate between the overhead in the text section and in the data section. We will analyze the overheads separately to differentiate between them according to their impact. During our program size measurements, we used the MSP430 implementation, which ws compiled with the GNU msp430-gcc compiler suite version 4.4.5.

We mentioned in Section 6.3.2.2 that the tinyVLIW8 soft-core processor does not use a generic ACL check. The checks are directly integrated, so that SPD-specific overhead is dominated. Hence, we evaluated the SPD-specific overhead for the SWUR firmware only.

Constant overhead

We measured the overhead by using the GNU program `msp430-size`, which is part of the GNU gcc compiler suite. The program can be used to analyze program files as well as object files, but requires that all sources are compiled with the GNU msp430-gcc compiler. Since we have implemented our security nucleus as a dedicated object file, we can measure its size separately and can compare it with the overall size of the tiny scale application. Table 7.9 gives an overview about the memory footprint of the Meetering app compiled for an MSP430F5438A with subset sizes of the langOS core, the Meetering app, and the security nucleus.

Table 7.9: Memory footprint of the Meetering app with subset sizes of the langOS core, the Meetering app, and the security nucleus.

Module	Text size (bytes)	%	Data size (bytes)	%
langOS library	21,812	83.8	1,030	80.8
Meetering app	3,748	14.4	238	18.7
libc	460	1.8	7	0.5
Security nucleus	2,424	8.5	2	0.2
	28,444	100.0	1,277	100.0

Since the memory protection nucleus is implemented as an extension of the MSPsim, it is similar to a hardware-based implementation, which does not cause a memory overhead on the software side. The security nucleus includes the MPU driver, the generic CDC, the generic ISR, and the DDT initialization. Hence, the constant overhead within the text section is 8.5 %. The constant overhead within the data sections (data and BSS) is smaller than the

program code overhead. It is only 0.2 %, because the overhead within the data sections is mostly caused by the SPD-specific data, which are not included here.

We can conclude that the constant overhead caused by the SN is small. In comparison to the available system resources of the MSP430F5438A, which is equipped with 256 kB non-volatile memory and 16 kB RAM, the resource utilization of the SN, 0.92 % in the text section and 0.01 % in the data sections, is negligible.

SPD-specific overhead

The SPD-specific overhead is caused by the CDCs integration and the data space description. The CDC integration causes an overhead in the text section as well as in the data section. In langOS each function call, which implements a CDC, is replaced by an assembler section as shown in Listing 7.1. The assembler section pushes the function code (FC) and the target SAID into the register R5, first. Afterwards, the function parameters are handled similarly to an ordinary function call. Finally, the generic CDC implementation of the SN is called.

Listing 7.1: CDC calls on an MSP430.

```
MOV R5, (#SAID << 8 ) | #FC

# function parameter handling

CALL #__wrap_fc
```

The overhead within the text section is caused by the MOV instruction only. Due to the fact that the number of parameters of the wrapped function is not altered, the function call is unchanged in its size. The SAID and the FC are stored in the temporary register that is treated as a fixed register. Hence, an additional stack operation to free the register can be avoided. We store the FC and the SAID in a single register to save an additional instruction on the caller's side. Thus, we can determine the exact overhead caused by the CDC by counting the number of CDC calls. Table 7.10 gives an overview about the CDCs of the Meetering app. Since each MOV instruction with a constant needs four bytes and the Meetering app includes 46 CDCs the overhead is 186 bytes only, which is 0.7 % of the Meetering app and 0.1 % of the available non-volatile memory of the MSP430F5438A.

Table 7.10: Overview about the Meetering app CDCs including the number of callers, the number of parameters, and the required data memory to store the ACL[3].

Software activity	Public functions	Caller	Calls	Parameter	ACL size (bytes)
STORAGE	2	2	2	0	12
CAPTURE	6	9	22	4	50
NETWORK	7	8	12	10	64
MAIN	5	5	5	4	38
RADIO	5	5	5	3	36
	25	29	46	21	200

The overhead within the data section is mainly caused by the SA context and the ACL. The SA context is used by the SN to store registers on a domain switch and the base pointers of

[3]Since the bootstrap SA does not have any public functions as well as any caller, it is not listed in the Table.

the function code tables. The size of an SA context is 48 bytes. The Meeting app uses six SA contexts since it defines six SAs. The size of an ACL entry depends on the number of callers and the number of function parameters. Table 7.10 gives an overview about the value of the Meetering app and summarizes the size of the ACL. The ACL consumes 200 bytes for 25 public functions, which increases the size of the data section of the Meetering app by 19 % and uses 1.2 % of the available memory of the MSP430F5438A.

Beside the ACL, the SN consumes data memory to store the DDT. We store the DDT in the main memory and make it accessible by the MPU. The size of a DDT entry is 80 bits. 40 bits are used to the store base address, the size, and the owner. The second 40 bits are used to store two ACL entries or three capability entries. Based on the number of data spaces, evaluated in Section 6.2.2.1, the DDT consumes 550 bytes within the data section, which is 43 % of the Meetering app memory and 3.4 % of the available MCU memory. Furthermore, the same amount of memory is required in the text section to store the statically initialized DDT content.

Table 7.11 summarized the overall memory footprint of the SN of the Meetering app. The results show that the main overhead is caused on the RAM resources. The SN consumes only 1.2 % of the available non-volatile memory, but 6.3 % of the MCU's RAM, which increases the RAM usage of the Meetering app by 82 %.

Table 7.11: Memory footprint of the security nucleus (SN) of the Meetering app (text 26,020 bytes / data 1,275 bytes) on an MSP430F5438A (text 262,144 bytes / data 16,384 bytes).

Component	Text size			Data size		
	bytes	App %	MCU %	bytes	App %	MCU %
SN	2,424	8.5	0.9	290	22.7	1.8
CDC	186	0.7	0.1	200	19.4	1.2
DDT	550	2.1	0.2	550	43.1	3.4
	3,160	12.1	1.2	1,040	81.6	6.3

The SWUR firmware does not feature a nucleus gate but the security nucleus causes an overhead by the integration of the CDCs and the ACL checks. Since the ACL checks are integrated statically in the functions, instead of using a generic call, the ACL is not stored in the RAM. The overhead is added within the text section only. Table 7.12 gives an overview about the CDCs implemented in the SWUR firmware.

Table 7.12: CDCs of the SWUR firmware. Each public function is called by one caller only.

SA	Public functions	Callers	Calls
CORE	2	2	2
TOTP	4	3	4
REG	2	2	2
SYMDEC	5	3	3

Each CDC requires eight additional instructions: two instructions to move FC and SAID into registers and six instructions for the ACL check. Hence, the overhead caused by the CDCs is 80 instructions, which increases the overall number of instructions by 9 %.

Beside the additional instructions for the CDCs, the SN on the tinyVLIW8 requires memory resources to initialize the DDT. The DDT memory is part of the hardware-based MPU, so that the firmware includes the DDT initialization only. As mentioned in Section 5.1.2.5, each DDT write operation includes five bytes, which have to be written into the registers of the MMIO interface. This operation needs nine instructions on the tinyVLIW8. Hence, the DDT initialization of the SWUR firmware requires 162 instructions (see Table 6.4, 18 DDT entries), which increases the overall number of instruction of the SWUR firmware by 18 %.

Due to the very small amount of RAM the SN for the tinyVLIW8 soft-core processor uses as less memory as possible. This optimization causes that the overhead within the program text is dominated. The CDCs and the DDT management consume 242 additional instructions, which increases the overall number of instructions by 27 % and consumes approximately 8 % of the available program memory[4]. The overhead in RAM and the overhead caused by the generic nucleus gate are irrelevant.

7.2.1.3 Memory overhead of the tiny hypervisor

The tiny hypervisor was implemented for the MSP430 MCU only. Due to the limited resources of a TSS an implementation of a tiny hypervisor for the tinyVLIW8 soft-core processor makes no sense. In the following, we will give a brief evaluation of the memory overhead of the tiny hypervisor on an MSP430. When assessing the results of the evaluation, it has to be considered that the overhead comes in addition to the overhead of the security nucleus.

The implementation of the tiny hypervisor and its overhead can be split into two parts:

- the size of the tiny hypervisor and the binary verifier and
- the binary instrumentation.

The tiny hypervisor is compiled to a library that is linked to the TSS application in a final step. The library includes the instruction emulation and the back end of the DDT management. Table 7.13 summarizes the memory footprint of the tiny hypervisor and the binary verifier.

Table 7.13: Memory footprint of the tiny hypervisor and the binary verifier.

Module	Text size (bytes)	Data size (bytes)
tiny hypervisor	960	4
binary verifier	426	0
	1,386	4

The overhead caused by the tiny hypervisor and the binary verifier is very low. It is less than 6 % of the text size of the original binary and only 0.5 % of the available non-volatile memory of the MSP430F5438A. A significant larger overhead is caused by the binary instrumentation. It depends on the firmware especially on the number of unsafe instructions. To give a rough quantitative evaluation of the overhead of the binary instrumentation, we instrumented the Meetering app. The results are summarized in Table 7.14.

[4]The program memory of the tinyVLIW8 soft-core processor can store 2048 VLIW instructions. We assumed a utilization of 75 % of the VLIW instructions to calculate the memory footprint overhead.

Table 7.14: Memory overhead caused by the binary instrumentation. The measurements are done on the object files, so that the results do not include libraries and the original binary size is smaller than in previous measurements.

Binary	Size (bytes)
Original	25,560
Instrumented	69,070

The instrumented binary is 170 % larger than the original one and in combination with the constant overhead added by the tiny hypervisor and the binary verifier the overhead is 178 %. Due to the significant overhead additional modifications might be necessary, e.g. jump labels might be to far to be jumped on directly and have to be modified accordingly. This overhead is not included yet.

Beside the program memory, the data memory is increased by a constant overhead only. This is less than 1 % and can be neglected. We must summarize that the binary instrumentation is very expensive and can be used only, if a firmware image leaves a significant amount of space within the non-volatile memory.

7.2.2 Performance evaluation

The general applicability of a security platform in TSSs is mainly driven by its design size. But the practicability is driven by the performance drawback of the platform. Hence, we will analyze the performance drawback of our security platform. Whereas the performance of common microprocessors is usually given by a value of million instructions per second (MIPS), an evaluation on deeply embedded systems is focused in clock cycles per operation. On each computer system each clock cycle consumes a specific amount of electricity and time. Whereas an increasing of the clock speed or the number of cores reduces the computation time, the power consumption of the system will be increased likewise, which is not practical for embedded systems. Hence, it is in particular important how many additional clock cycles are consumed by an extension. Therefore, we will introduce an extendable cycle accurate simulator (CAS) to count the clock cycles of program sections on an MCU platform.

In the following, we will evaluate the performance drawbacks of the most critical components of our security platform. But we shortened this evaluation because of large components of our platform have currently very rough prototype status and were never optimized in performance.

7.2.2.1 Hybrid simulation environment (HSE)

The hybrid simulation environment (HSE) combines a CAS for the MSP430 written in Java and with a SystemC kernel, which allows the description of peripheral units at a reasonable abstraction level.

Introduction to the MSPsim

The MSPsim is a cycle accurate simulator (CAS) that simulates the target's ISA at the instruction level and includes the execution time of each instruction. The ISA of an MSP430 can be

simulated by the MSPsim [EDF+08]. The MSPsim, developed by the SICS, is completely written in Java and is able to run unmodified target platform firmwares. The Java programming language makes the simulator very attractive for an early stage testing of software components. Furthermore, the behavior of new peripheral components can be described as a Java model as well. These components can be easily integrated in the MSPsim by its well-defined memory mapping interface. The combination of a component behavior description and a processing core simplifies system testing and evaluation in a significant manner.

Coupling MSPsim with synthesizable hardware models

Although the implementation of a behavior model of a new hardware component in Java is quite comfortable an additional implementation step to port the design to a synthesizable hardware model is always necessary. We used the advantages in a hybrid simulation environment (HSE) for an MSP430 [SMV+11]. We combined the CAS MSPsim with a SystemC simulation kernel. As shown in Figure 7.3 the Java-based MSPsim is connected to the SystemC environment via the memory bus interface. Our work was focused on an easy integration of new memory mapped components as well as the developed MPU.

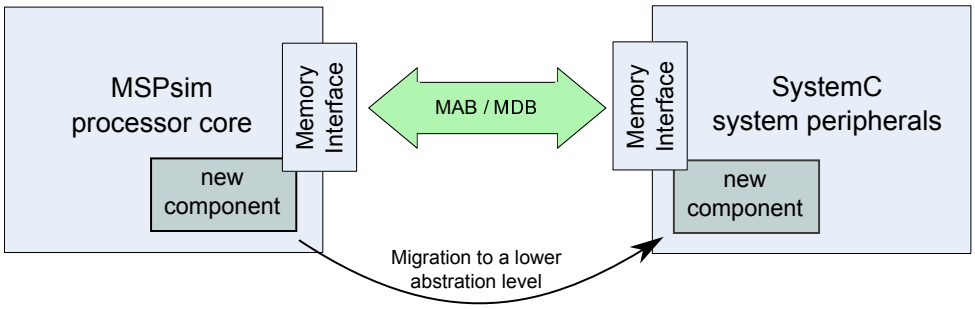

Fig. 7.3: The hybrid simulation environment for the MSP430 allows a migration of new peripheral components to the SystemC kernel. The MSPsim exports the MAB and the MDB to connect components within the SystemC kernel.

For connecting the SystemC environment to the MSPsim a communication class had to be added, which implements the MAB and the MDB with the signals *read*, *write* and *chip select*. Beside the memory interface the SystemC extension supports interrupts as well. An interrupt can be generated by any SystemC component and will be forwarded via an asynchronous interface.

Integration of the MPU

Following the integration of the tailor-made MPU in real systems, modifications on the HSE memory interface became necessary. As mentioned in Section 5.1.2 the MPU must be placed between the processor core, the peripherals, and the memory. But the original HSE430 supports peripheral components implemented in the Java CAS as well as parts of the SystemC extension. Due to the MPU's SystemC implementation any access to Java components is covered by the MPU unless the access is routed to the SystemC core first. To avoid such a complex setup only SystemC peripherals are used during evaluation. For a benchmark it is important that all peripherals are placed behind the MPU.

For a fully-functional integration of the MPU two additional signals between the MSPsim and the SystemC environment had to be implemented. Due to the shared memory of the Von-

Neumann-Architecture of the MSP430 an additional execute signal is necessary. The signal must be routed from the processor core's execution unit to the MPU and signals an instruction fetch. The HSE was extended to provide a similar signal. Each memory request includes a stage flag that signals the MPU the current execution stage of the processor core that triggers the request. In case of a DLB-miss the processor core must be stalled until the match operation within the segment table is finished. The stalling is implemented by an additional signal routed from the MPU to the processor core.

7.2.2.2 Performance evaluation of critical components

In our performance evaluation we analyzed the impact of the MPU, the capability-based CDC, and the tiny hypervisor. We ignored the static overhead caused by the DDT initialization or the binary verifier. Their overhead is caused only at boot-strap and does not have any impact on the run-time performance of the system when executing a given task.

We focused our evaluation on the MSP430-based platform. The performance overhead on the tinyVLIW8 soft-core processor is mostly static and an evaluation is quite simple. Since the MCU is extended by a CAM-based MPU, a performance drawback caused by the DDT look-up does not exist. Each look-up is performed in parallel to the memory access. Furthermore, a CDC is coded in a very short SA switch, where we integrated an optimized inline ACL check, so that the whole CDC is coded by six instructions. Finally, a tiny hypervisor was never implemented. Hence, we can summarize that the performance drawback on the platform is only a few clock cycles on each CDC.

In the following, we give evaluations for the three components on MSP430-based systems. The MSP430 MPU is equipped with a DLB-based DDT, which causes a run-time overhead.

Impact of the DLB-based MPU

The run-time overhead caused by the DLB-based MPU must be differentiated into an DLB-hit and a DLB-miss. In case of a DLB-hit the overhead is zero due to the fact that theDLB check is performed in parallel with the memory access. The DLB-hit rate depends mainly on the structure of the program and is similar to processor's cache. It depends on the temporal and the spatial locality of the program code. Hence, a general statement of the impact of the DLB-based MPU cannot be given.

We analyzed the number of DLB look-ups necessary for a single instruction of the MSP430. Without a deeper analysis the number of DDT look-ups will be equal to the number of 16-bit words of an instruction. But since each instruction is placed in a single data space, the load of the first 16-bit word will raise one DDT look-up only. The addresses of the following 16-bit words will match with the DLB entry last loaded and will raise a DLB-hit. Table 7.15 gives an example of a double operand instruction of an MSP430.

Beside the instruction data, each instruction may cause an additional data access. In case of a double operand instruction on an MSP430, both operands may point to different data spaces and may cause individual DDT look-ups. Hence, as a worst case, three individual DDT look-ups might be necessary. In the best case all the needed data spaces are already loaded into the DLB and no additional overhead is caused by the MPU.

The overhead of each DDT look-up depends on its implementation. The current DDT look-up is implemented as a sequential search. Hence, the number of clock cycles of a DLB-look-up

Table 7.15: Run-time estimation, divided in DLB-hits and DLB-misses, of a double-operand instruction for four different addressing modes.

Instruction	Size (16-bit words)	Cycles		
		original	DLB-hit	DLB-miss
mov(Rn, Rm)	1	1	1	1 + $lookup$
mov(@Rn, Rm)	1	2	2	1 + $2 \times lookup$
mov(x(Rn), Rm)	2	3	3	2 + $2 \times lookup$
mov(x(Rn), ADDR)	3	6	6	6 + $3 \times lookup$

$lookup$ is in average $\frac{(dsn+1)}{2} \times i_{match}$, where dsn is the number of data spaces and i_{match} is the number of clock cycles per entry match.

Performance drawback of capability-based CDC

The MSP430 ISA supports the `call` instruction to enter a subroutine. The instruction needs five clock cycles. The subroutine return is emulated by the instruction `mov @sp+, pc`, which needs four clock cycles. In case of a CDC the `call`-instruction is replaced and has to perform a context-switch, parameter marshalling, and an ACL check. In case of a CDC-return, the previous context must be restored and parameters must be marshalled optionally.

Table 7.16 shows the values of a CDC call on an MSP430. Since parameter marshalling and ACL check are optional, we measured four different cases.

Table 7.16: Clock cycles required for a CDC.

CDC	Marshalling	ACL	Clock cycles
func1(void)	-	-	354
func2(char *buf)	x	-	1129
func3(void)	-	x	396
func4(char *buf)	x	x	1171

In comparison to the nine clock cycles of an unmodified subroutine call the current overhead is huge. Especially parameter marshalling is very expensive. The current implementation includes a data space look-up based on a given address. This look-up needs statically 31 cycles and 104 cycles per data space. Our example traverses three data spaces. In a worst case the look-up might consume 11,440 clock cycles. The current implementation does not include any performance optimization, so that improvements are possible. But we have to consider that a CDC is expensive and should be used with care. The overhead has to be taken into account, when designing the software, so that a demanded task can be performed with a reasonable number of CDCs.

Performance of the tiny hypervisor

We have shown in Section 6.1.3.2 that the emulation of a virtual instruction is split in two parts. The first part includes the instruction fetch and the instruction emulation. The second part includes the access control. Both parts must be executed on each emulated instruction. We implemented the first part in assembler to have control over the used register. Performance optimizations are not done yet. We identified that the detection of the register used

by the virtual instruction is very expensive. Currently we implemented a simple search algorithm, which starts with the lowest possible register and ends with register R15. In case of register R15 all registers have to be compared, which results in a large number of additional instructions.

The DLB look-up is implemented in standard C. In case of an DLB hit only 15 instructions are necessary. But in case of an DLB-miss a DDT look-up is necessary. We proposed a guarded DDT, which is a trade-off between memory space and performance. The number of instructions for a look-up depends on the size and the content of the DDT. Hence, a general result for a TSS application cannot be given.

7.2.3 Comparison with state-of-the-art of technology

A comparison of performance and resource utilization with state-of-the-art of technology is difficult to give. The Infineon TriCore MCU used by the KESO approach and the MSP430 FR57xx family are silicon devices. Measurements for an MPU caused drawback are not given. The MMP was not implemented in hardware so that real measurements are also not available. Hence, we can compare the resource utilization of our approach with the UMPU, the Sancus approach, and the Nios II soft-core processor only.

The UMPU was integrated in an AVR ATmega 8-bit MCU. The authors count the gates to specify the hardware size. The unmodified MCU includes 23,104 gates and the security enhanced version uses 33,855 gates, which is an overhead of 46 %. The approach uses the system memory to store the memory map table. Hence, the overhead must be compared with a DLB-based design. In that case our design causes an overhead of 33 % only. The Sancus approach uses a combinatorial look-up scheme as presented by our CAM-based design. The overhead specified by the authors of Sancus is 307 LUTs and 213 registers for each protection domain. In case of implementing 32 protection domains, the design uses 9,824 LUTs and 6,816 registers, which is much more expensive than our approach. Furthermore, we can compare our design with the Nios II MPU provided by Altera. The overhead of a Nios II MPU specified by Altera is 600 LUTs [Alt15], which is quite similar to our DLB-based approach.

The memory footprint caused by alternative approaches is given by Kumar for the UMPU and Stilkerich for KESO. The UMPU proposed by Kumar has a data memory overhead of 6.3 % and a code overhead of 2.8 %. Stilkerich specifies an overhead of less than 1 % in data size and 18 % in code size. The overhead caused by our approach is mainly driven by the size of the DDT. For the Meetering example app the memory overhead is 6.3 % of the available MCU resources. Since both approaches do not provide a large number of data spaces and the exact numbers is unclear a direct comparison cannot be given.

Measured values about the performance overhead are given by the KESO and UMPU approach as well. A CDC in KESO differs with from 192 to 457 clock cycles. The number depends on the chosen level of memory protection. Compared to a regular function call the costs increases to a factor of 17 for the invocation and the return. The UMPU approach requires only 11 additional CPU cycles for a CDC. The measured values include only a domain switch and return. The operation is handled mostly in hardware. In comparison to our platform, both approaches do not feature an ACL check, parameter marshalling and stack clearing, which makes our approach much more expensive. But the missing three features

are mandatory for a secure system. Since KESO and UMPU are focused on safety, the features have not to be provided their safety goals.

We can summarize that the performance and the resource utilization of our hardware-based isolation of software activities is very difficult to compare with state-of-the-art of technology. But considering the fact that only our approach enforces security the achieved results are reasonable in a direct lineup.

CHAPTER 8

Conclusion

In this thesis, I presented a platform that enables a secure isolation for software activities on tiny scale systems (TSSs). The lack of resource isolation makes TSSs vulnerable for a broad variety of malicious software. The main goal was to provide basic principles to build secure systems as already established on commodity server, desktop, and in particular mobile computer systems. To cope with the restricted resources of TSSs, the developed platform takes the special characteristics of these systems into account. Hence, my approach is based on a co-design process that includes hardware, compile-time, and run-time parts, so that performance drawbacks at run-time and extensive memory requirements can be mostly avoided.

In this final Chapter, I will summarize the presented work and will emphasize my main contributions. In the following, I will describe limitations and future activities, which were out of the scope of my thesis.

8.1 Summary

The work was motivated by the upcoming, ubiquitous availability of information systems, which has changed our modern society significantly. The proceeding penetration and the rapidly increasing interconnection of embedded systems move them into the focus of security investigations. Physical fences and administration guidelines are no longer an adequate instrument to protect deeply embedded systems. Recent news have shown that major critical systems, such as modern automobiles, industrial plants, as well as power stations, are targets of malicious adversaries. The current and further upcoming challenges of embedded systems ask for a platform that enables the opportunity to build secure systems. Such a platform must ensure that software activities have access only to those resources, which are assigned to them. Motivated by this issue, I presented a design and a prove of concept implementation of a platform that makes such a demanded secure isolation of software activities possible.

Given by the fact that the term embedded systems covers a very broad variety of systems, I started my thesis with a definition of tiny scale systems and software activities, which are in the focus of my work. In the following, I sketched vulnerabilities and weaknesses of this class of systems and pointed out that in particular local attacks are highly critical. In the following, I introduced technologies to build modern, secure systems, which are well-established on commodity computer systems. A further goal of this thesis was to analyze the applicability of these technologies for TSSs. As a fundamental core of my thesis I presented the concept of a platform for security enhanced TSSs, which is based on four basic principles: tailor-made data spaces, software flow integrity, a trustworthy instance, and a fine-grained access control. To prove the applicability of my platform, I described its assembling on two real system architectures and a port of two real TSS applications. The benefit of the presented platform

is mainly measured by its security gain. Hence, I added a qualitative security evaluation that emphasizes the countermeasures based on the four basic platform principles to close local weaknesses and threats. My evaluation is completed by a quantitative comparison of my platform with state-of-the-art of technology.

8.2 Contributions and limitations

The primary goal of my work was the design, the implementation, and the evaluation of a platform for security enhanced tiny scale systems (TSSs). The platform aims to reduce the remaining risks caused by local weaknesses and threats of deeply embedded systems. Since typical applications of TSSs do not include a proper isolation of resources, a common OSs, and user-centered design, well-known technologies of commodity operating systems cannot be used without adaptions. Hence, the major contribution of my work was to make well-established security technologies of common desktop, server and powerful mobile devices applicable for TSSs. The core components of this major contribution can be summarized as follows:

A tailor-made MPU was designed and assembled in hardware for two real system architectures that cover both fundamental computer architectures: the von-Neumann architecture and the Harvard architecture. In addition, a software-based approach applicable for off-the-self MCUs was presented. The MPU provides a fine-grained segmentation that covers single peripheral registers as well as large memory sections to take all characteristics of embedded systems into account.

An adapted role-based access control (RBAC) scheme for TSS applications was defined to provide a trustworthy software flow integrity. An application-specific security policy definition (SPD) assigns activities, roles and operations of the RBAC model to the elements of a TSS and defines access control as well as information flows between them.

A fast domain switching is provided by the MPU to implement a kernel-less system. Especially TSSs ask for an efficient implementation of isolated resources. Given by the fact that TSSs are very often used in real-time applications, each additional isolation must be applicable with a minimal drawback in performance and resources. Hence, a fast domain switch is mandatory for an efficient enforcement of a fine-grained resource isolation.

A compile- and run-time co-design process is given that takes the characteristics of TSS applications into account. The proposed process makes use of pre-defined and static elements of a TSS application to integrate enhanced security mechanisms at compile-time. The process enables an enforcement of complex security schemes at run-time with reasonable drawbacks in performance and resource utilization.

The enforcement of a secure isolation requires a trusted computing base (TCB) implemented in software or in hardware. The selection of the capabilities provided by the TCB influences

the system's flexibility significantly. It is always a trade-off that induces limitations either in flexibility or in security. I have focused the design of my platform on security, so that I had to have limited the provided run-time flexibility. Common features as loadable program sections or dynamic memory management may be possible with significant overhead only. These features were not considered during the platform's design process. Furthermore, I must consider that the imposing of data spaces is the most critical weakness of my approach. The protection of the MPU register solves the problem but restricts the flexibility of the system in a significant manner. Hence, the implementation of a two-way hand-shake may be a reasonable option. But further investigations are necessary to provide a more flexible scheme with the same or a higher level of security.

I gave a quantitative cost evaluation that has illustrated that security cannot be provided without a significant overhead. Although the current implementation is not optimized in performance and resource utilization, I have shown that the platform can be enforced on real application and real system architectures. Nevertheless, an implementation of more complex and real applications would require further improvements.

8.3 Future activities

Nearby, future activities must be improvements in resource utilization and performance. I have already presented some ideas of appropriate mechanisms within the previous Chapters. In the following, I will conclude this thesis with some ideas of future activities beside performance and resource utilization improvements.

8.3.1 Completing the security platform

The in the following sketched approaches include components that either could not be realized within the scope of this thesis or are a direct outcome of the evaluation of my platform.

Extended RBAC support in langOS

I introduced the concept of langOS interfaces. In my current implementation I added source code annotations into both C sources and interface files. A more clear structure would be given if annotations would be limited to interface files. This approach would require that the interface support of langOS must be extended and all source files of langOS make use of interfaces. Furthermore, an improved adaptation of users and sessions would simplify the enforcement of the RBAC model significantly.

The current implementation of langOS is focused on an efficient use of resources. I have shown that within the context of security a shared use of resources increases the TCB significantly. Hence, in a further task a tight symbiosis of langOS and my security platform has to enforced. Such a closely combination of an OS library and my platform security is mandatory to build systems with high security demands. To the best of my knowledge no state-of-the-art of technology provides a comparable system.

No-return operating system

A forceful continuation of my kernel-less system architecture would be given by a stream-like operating system. As mentioned in Section 4.2.2.2, the enforcement of a secure isolation of SAs requires a very careful handling of the program stack, so that the integrity of the control flow can be guaranteed on the return path. I am convinced that the secure isolation of SAs in TSS applications can be simplified significantly without losing functionality by removing most of these return paths. Especially event-driven OSs work like a stream processor, on which an event is handled in a pipeline of software modules, so that the concluding return path is mostly useless.

My extended compilation model introduces CDCs as a replacement of common function calls. The implementation of a no-return OS would require an adaptation of langOS. Furthermore, the source code annotations must be able to differentiate between synchronous and asynchronous CDCs. Only in case of a synchronous CDC a return must be considered. Otherwise a return on an asynchronous CDC can be forbidden, which would increase the system's security as well.

CoMet-based SPB optimization

I introduced the configurable compiler suite CoMet in Section 6.3.3. The compiler suite was extended by us to enforce the security policy compiler (SPC) in programs for the tinyVLIW8 soft-core processor. But the CoMet suite was initially developed to configure the tool chain on each transformation step to simplify a hardware-software co-design process. Hence, after each transformation step the resulting system can be simulated to check achieved results against the requirements.

The simulation capability would allow us to analyze the impact of an application-specific security policy definition (SPD) in detail. In a further extension of the compiler suite additional modules can be implemented. These modules will give us additional information about the expected performance drawback and the memory overhead or can provide a memory heat-map that shows data clusters. The information can be used to adapt the SPD to find an ideal trade-off between security, performance, and resource utilization. The simulation instrument would give a developer a direct feed-back about his application and his security definition, so that the compile-time/run-time co-design process can be further improved.

We used the CoMet suite to compile tinyVLIW8 programs. A support for an additional architecture is not given yet. Due to the fact that the tinyVLIW8 soft-core processor is focused on small control tasks and does not have an active community yet, a support of the MSP430 platform is mandatory.

Hardware-based MPU for an MSP430

I sketched a hardware-based MPU for an MSP430 soft-core processor. But as mentioned in Section 7.2.1 an implementation of the MPU is still missing. The MSP430 MCU family is widely used within the community of embedded systems and especially in WSNs. Therefore, a port of our concept to this soft-core processor or any compatible device might be aimed.

The NEO430 soft-core processor was published on the OpenCores platform in October 2015 [Nol15]. The design was already proven on an Altera Cyclone IV FPGA, which is similar to

the FPGA used on the IHPstack sensor node. It is the third implementation of an MSP430-compatible soft-core processor beside the IHP430X and the MSP430 provided by Girard. All these soft-cores are available in source. But in contrast to the previously available implementation the NEO430 overcomes the limitations of the IHP430X and the Girard MSP430. The NEO430 is written in VHDL and features a synchronous design. Both simplify the integration of my MPU significantly.

The implementation of a hardware-based MPU for the MSP430 will complete our enhanced security platform for MSP430-based TSSs. After finishing all these activities we will support a common processor core, provide a hardware-based MPU, a configurable compiler suite, and a security-focused OS library. The publication of all these components to the embedded system community as public domain, is seen as an upcoming goal in the near future.

8.3.2 Strong security platform

I focused my security platform on the prevention of local weaknesses and threats. But especially embedded systems with their exposed position are vulnerable for tamper attacks. An all-embracing security platform must prevent this type of attacks as well. In my publication of an implementation of an intrinsic code attestation for embedded devices, I proposed a further security mechanism to protect deeply embedded systems. A combination of both approaches would provide a strong security platform. It allows an execution of encrypted program sections so that software-based security mechanisms cannot be manipulated.

The approach lacks currently of an encryption of the data section. It has to be integrated to protect security critical data, such as data space information and the ACLs. A fully encrypted system with an fine-grained isolation of software activities would reduce the remaining risk for local and tamper attacks significantly. I am convinced that such a system's platform will provide the demanded security for current and future application scenarios of deeply embedded systems.

Acronyms

ACL	access control list
AOP	aspect-oriented programming
API	application programming interface
ASIC	application specific integrated circuit
ASIP	application-specific instruction-set processor
AUTOSAR	automotive open system architecture
BEV	battery electric vehicle
CAM	content-addressable memory
CAS	cycle accurate simulator
CCS	Code Composer Studio
CDC	cross-domain call
CFG	control flow graph
CFI	control flow integrity
CI	critical infrastructure
CPU	central processing unit
CPS	cyber-physical system
DAC	discretionary access control
DCO	digitally controlled oscillator
DDT	data space descriptor table
DLB	data space lookaside buffer
DoS	denial of service
DSID	data space identifier
DVM	dynamically extensible virtual machine
ECC	error-correcting code
FC	function code
FMC	flash memory controller
FPGA	field programmable gate array
GCC	GNU compiler collection
GCF	global configuration file
GIE	global interrupt enable
GLT	group lookup table
GPIO	general-purpose input/output
HAL	hardware abstraction layer
HMAC	keyed-hash message authentication code
HSE	hybrid simulation environment
HOTP	hash-based one-time password
IDL	interface definition language
IVT	interrupt vector table
IC	integrated circuit
IoT	internet of things
IPC	inter-process communication
IP	intellectual property

IBMAC	instruction based memory access control
ISA	instruction set architecture
ISR	interrupt service routine
ISS	instruction set simulator
IT	information technology
JVM	Java virtual machine
LC	logic cell
LoC	lines of code
LUT	look-up table
MAC	mandatory access control
MACM	memory access control matrix
MAB	memory address bus
MCU	micro controller unit
MDB	memory data bus
MIT	Massachusetts Institute of Technology
MLS	multi-level security
MMIO	memory mapped input/output
MMP	Mondriaan memory protection
MMU	memory management unit
MPU	memory protection unit
MSB	most significant bit
MT	multi-thread
NIST	National Institute of Standards and Technology
OS	operating system
PC	personal computer
RAM	random access memory
RBAC	role-based access control
RISC	reduced instruction set computer
RLB	rights lookaside buffer
ROM	read only memory
ROP	return-oriented programming
RPC	remote procedure call
SA	software activity
SAID	software activity identifier
SFR	special function register
SHA1	Secure Hash Algorithmus 1
SHP	single hop protocol
SICS	Swedish Institute of Computer Science
SIT	size in 2^n
SFI	software-based fault isolation
SLT	segment lookup table
SN	security nucleus
SPD	security policy definition
SPB	security policy book
SPC	security policy compiler
SPI	serial peripheral interface
SWUR	secure wake-up receiver
TCB	trusted computing base
TCG	trusted computing group

TI	Texas Instruments
TLB	translation lookaside buffer
TOTP	time-based one-time password
TPM	trusted platform module
T-RBAC	task-role based access control
TSS	tiny scale system
UART	Universal Asynchronous Receiver Transmitter
UMPU	micro memory protection unit
VHDL	very high speed integrated circuit hardware description language
VIS	virtual instruction set
VLIW	very large instruction word
VM	virtual machine
VMM	virtual machine monitor
WDT	watchdog timer
WSN	wireless sensor network
WUR	wake-up receiver

List of Figures

List of Tables

List of Listings

Bibliography

[ABB+86] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer Conference*, pages 93–113, Altanta, GA, USA, June 1986.

[ABEL05a] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communication Security (CCS)*, number MSR-TR-2005-18, pages 340–353, Alexandria, VA, November 2005.

[ABEL05b] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A theory of secure control-flow. In *International Conference on Formal Engineering Methods (ICFEM)*, number MSR-TR-2005-17, pages 111–124, Manchester, UK, November 2005. Springer-Verlag.

[ABS94] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 290–301, New York, NY, USA, 1994. ACM.

[AFH+06] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing process isolation. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, MSPC '06, pages 1–10, New York, NY, USA, 2006. ACM.

[AK96] Ross Anderson and Markus Kuhn. Tamper Resistance: A Cautionary Note. In *Proceedings of the 2Nd Conference on Proceedings of the Second USENIX Workshop on Electronic Commerce*, volume 2 of *WOEC'96*, pages 1–1, Berkeley, CA, USA, 1996. USENIX Association.

[AK98] Ross J. Anderson and Markus G. Kuhn. Low Cost Attacks on Tamper Resistant Devices. In *Proceedings of the 5th International Workshop on Security Protocols*, pages 125–136, London, UK, UK, 1998. Springer-Verlag.

[ALE+01] Mohit Aron, Jochen Liedtke, Kevin Elphinstone, Yoonho Park, Trent Jaeger, and Luke Deller. The Sawmill Framework for Virtual Memory Diversity. In *Proceedings of the 6th Australasian Conference on Computer Systems Architecture*, ACSAC '01, pages 3–10, Washington, DC, USA, 2001. IEEE Computer Society.

[Ale05] Steven Alexander. Defeating Compiler-level Buffer Overflow Protection. *;login: The USENIX Magazine*, 30(3), June 2005.

[Alt12] Altera. Cyclone III Device Handbook, 2012. `http://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyc3/cyclone3_handbook.pdf`.

[Alt15] Altera. Nios II Core Implementation Details, 2015. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/nios2/n2cpu_nii51015.pdf.

[And13] Nate Anderson. Meet the men who sky on woman through their webcams. Website, March 2013. http://arstechnica.com/tech-policy/2013/03/rat-breeders-meet-the-men-who-spy-on-women-through-their-webcams/.

[ARRJ06] Divya Arora, Anand Raghunathan, Srivaths Ravi, and Niraj K. Jha. Architectural Support for Safe Software Execution on Embedded Processors. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '06, pages 106–111, New York, NY, USA, 2006. ACM.

[ASE+08] Faisal Aslam, Christian Schindelhauer, Gidon Ernst, Damian Spyra, Jan Meyer, and Mohannad Zalloom. Introducing takatuka: A java virtualmachine for motes. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 399–400, New York, NY, USA, 2008. ACM.

[Ayc06] John Aycock. *Computer Viruses and Malware (Advances in Information Security)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[BA97] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, June 1997.

[Bar07] Richard Barry. Compiler verification for safety-critical applications. *embedded systems design europe*, June 2007.

[Bar10] Richard Barry. *Using the FreeRTOS Real Time Kernel - a Practical Guide*. FreeRTOS Tutorial Books. freeRTOS, January 2010.

[BBD06] Alexander Becher, Zinaida Benenson, and Maximillian Dornseif. Tampering with motes: Real-world physical attacks on wireless sensor networks. In *Proceedings of the Third International Conference on Security in Pervasive Computing*, SPC'06, pages 104–118, Berlin, Heidelberg, 2006. Springer-Verlag.

[BC13] Matthew Brocker and Stephen Checkoway. iseeyou: Disabling the macbook webcam indicator led. 2013.

[BCD+05] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. *Mob. Netw. Appl.*, 10(4):563–579, August 2005.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[BDJ88] U. Beyer, Heinrichs D., and Liedtke J. Dataspaces in l3. *Mini and Microcomputers and Their Applications*, 1988.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC'05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[Ber12] Erik Bergmann. *Entwurf und Implementierung einer werkzeugbasierten Aufteilung von Sensorknoten-Software in isolierte Adressräume*. Master thesis, Brandenburgisch Technische Universität Cottbus, 2012.

[BFH+92] Allen C. Bomberger, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The keykos nanokernel architecture. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 95–112, Berkeley, CA, USA, 1992.

[BHG+13] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt. RIOT OS: Towards an OS for the Internet of Things. In *Proceedings of IEEE Conference on Computer Communications Workshops*, INFOCOM WKSHPS 2013, pages 79–80, April 2013.

[BHR+06] Rahul Balani, Chih-Chieh Han, Ram Kumar Rengaswamy, Ilias Tsigkogiannis, and Mani Srivastava. Multi-level software reconfiguration for sensor networks. In *Proceedings of the 6th ACM IEEE International Conference on Embedded Software*, EMSOFT '06, pages 112–121, New York, NY, USA, 2006. ACM.

[BL73] D. Elliott Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical report, The MITRE Corp., Bedford, MA, USA, May 1973.

[BLC09] Niels Brouwers, Koen Langendoen, and Peter Corke. Darjeeling, a feature-rich vm for the resource poor. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 169–182, New York, NY, USA, 2009. ACM.

[Bor15] Borland. Devpartner studio professional edition 11.03. Datasheet, 215. `http://www.borland.com/Borland/media/Resources/Data%20sheets/BDS-DevPartner-Studio-Pro-Ed_11-3.pdf`.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 267–283, New York, NY, USA, 1995. ACM.

[BZ11] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 213–223, Washington, DC, USA, 2011. IEEE Computer Society.

[CAE+07] 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, pages 205–218, New York, NY, USA, 2007. ACM.

[CCJ+07] Hojung Cha, Sukwon Choi, Inuk Jung, Hyoseung Kim, Hyojeong Shin, Jaehyun Yoo, and Chanmin Yoon. RETOS: resilient, expandable, and threaded operating system for wireless sensor networks. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 148–157, New York, NY, USA, 2007. ACM.

[CDG+92] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *SIGPLAN Not.*, 27(8):15–42, August 1992.

[CGK+09] Yang Chen, Omprakash Gnawali, Maria Kazandjieva, Philip Levis, and John Regehr. Surviving sensor network software faults. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 235–246, New York, NY, USA, 2009. ACM.

[CHA+07] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *Proceedings of the 16th European Conference on Programming*, ESOP'07, pages 520–535, Berlin, Heidelberg, 2007. Springer-Verlag.

[CKD94] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 319–327, New York, NY, USA, 1994. ACM.

[Com07] Wikimedia Commons. Privilege rings for the x86 available in protected mode, 2007. `http://upload.wikimedia.org/wikipedia/commons/2/2f/Priv_rings.svg`.

[Con63] Melvin E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, July 1963.

[Con15] Jeremy Condit. Deputy. Website, 2015. `http://web.stanford.edu/class/cs295/asgns/asgn5/www/` [Stand: 04.08.2015].

[Cor63] A. Corneretto. Associative memories: A many-pronged design effort. *Electronic Design*, Vol 2:40–55, Feburary 1963.

[Cor06] Moteiv Corporation. Tmote Sky Ultra low power IEEE 802.15.4 compliant wireless sensor module. Datasheet, June 2006. `http://www.eecs.harvard.edu/~konrad/projects/shimmer/references/tmote-sky-datasheet.pdf` [Stand: 06.08.2015].

[CPM+98] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[CSC72] F. J. Corbató, J. H. Saltzer, and C. T. Clingen. Multics: The First Seven Years. In *Proceedings of the Spring Joint Computer Conference*, AFIPS '72, pages 571–583, New York, NY, USA, May 1972. ACM.

[DBMZ08] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. *SIGPLAN Not.*, 43(3):103–114, March 2008.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[DEO09] Ilker Demirkol, Cem Ersoy, and Ertan Onur. Wake-up receivers for wireless sensor networks: benefits and challenges. *Wireless Communications*, 16(4):88–96, August 2009.

[Des97] Solar Designer. return-to-libc attack. *Bugtraq mailing list*, August 1997.

[DGV04] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. *Local Computer Networks, Annual IEEE Conference on*, 0:455–462, 2004.

[DKAL05] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without garbage collection for embedded applications. *j-TECS*, 4(1):73–111, February 2005.

[EAV+06] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association.

[EDF+08] Joakim Eriksson, Adam Dunkels, Niclas Finne, Fredrik Österlind, Thiemo Voigt, and Nicolas Tsiftes. Demo abstract: MSPsim - an extensible simulator for MSP430-equippedsensor boards. In *Proceedings of the 5th European Conference on Wireless Sensor Networks*, EWSN '08, Bologna, Italy, January 2008.

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[ETFP12] Karim Eldefrawy, Gene Tsudik, AurÃ©lien Francillon, and Daniele Perito. Smart: Secure and minimal architecture for (establishing dynamic) root of trust. In *NDSS*. The Internet Society, 2012.

[Fab74] R. S. Fabry. Capability-based addressing. *Commun. ACM*, 17(7):403–412, July 1974.

[FC08] 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, pages 15–26, New York, NY, USA, 2008. ACM.

[FD92] D F Ferraiolo and D.R.Kuhn. Role-Based Access Control. In *Proc. of the 15th National Computer Security Conference*, pages 554–563, 1992.

[For04] M. Forster. Running Leon2 on the Altera Nios Development Board, Cyclone Edition. Website, 2004. `http://www.mdforster.pwp.blueyonder.co.uk/LeonCyclone.html` [Stand 03.06.2015].

[FPC09] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending embedded systems against control flow attacks. In *Proceedings of the first ACM workshop on Secure execution of untrusted code*, SecuCode'09, pages 19–26, New York, NY, USA, 2009. ACM.

[Fra10] Fraunhofer Institute for Integrated Circuits (IIS). Ultra low-current wakeup receiver, March 2010.

[Gai03] Jiri Gaisler. *LEON2 Processor User's Manual*. Aeroflex Gaisler AB, Goteborg, Sweden, 1.0.24 edition, 2003.

[GF09] Travis Goodspeed and Aurélien Francillon. Half-blind Attacks: Mask ROM Bootloaders Are Dangerous. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT'09, pages 6–6, Berkeley, CA, USA, 2009. USENIX Association.

[GGD+09] Saugata Ghose, Latoya Gilgeous, Polina Dudnik, Aneesh Aggarwal, and Corey Waxman. Architectural support for low overhead detection of memory violations. In *Proceedings of the Design, Automation and Test in Europe*, DATE '09, pages 652–657, Nice, France, April 2009.

[Gir10] Olivier Girard. openMSP430. Website, 2010. `http://opencores.org/project,openmsp430`.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[GLD00] Steve Guccione, Delon Levi, and Daniel Downs. A reconfigurable content addressable memory. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, IPDPS '00, pages 882–889, London, UK, UK, 2000. Springer-Verlag.

[GLvB+03] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 1–11. ACM, 2003.

[Goo07] Travis Goodspeed. Stack overlow exploits of 802.15.4 wireless sensors. Toorcon, San Diego, CA, USA, September 2007.

[Goo08] Travis Goodspeed. A side-channel timing attack of the MSP430. Black Hat, Las Vegas, NV, USA, August 2008.

[Gos95] James Gosling. Java intermediate bytecodes: Acm sigplan workshop on intermediate representations (ir'95). In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, IR '95, pages 111–118, San Francisco, CA, USA, 1995. ACM.

[Grä10] Hagen Grätz. 16 bit Microcontroller Core IPMS_430, 2010. `http://www.ipms.fraunhofer.de/content/dam/ipms/common/products/WMS/Cores/ipms430-e.pdf`.

[GS06] 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*, SenSys '06, pages 1–14, New York, NY, USA, 2006. ACM.

[Han70] Per Brinch Hansen. The Nucleus of a Multiprogramming System. *Commun. ACM*, 13(4):238–241, April 1970.

[Här02] Hermann Härtig. Security architectures revisited. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW'10, pages 16–23, New York, NY, USA, 2002. ACM.

[Hei08] Gernot Heiser. The Role of Virtualization in Embedded Systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, IIES '08, pages 11–16, New York, NY, USA, 2008. ACM.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The Performance of Micro-Kernel-Based Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, number 16 in SOSP 1997, Saint-Malo, France, October 1997. ACM.

[Hil92] Dan Hildebrand. An architectural overview of qnx. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Berkeley, CA, USA, 1992. USENIX Association.

[HJ92] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proceeding of the USENIX Winter 1992 Technical Conference*, pages 125–138, San Francisco, CA, US, January 1992.

[HKS+05] 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, pages 163–176, New York, NY, USA, 2005. ACM.

[HLP+00] Andreas Haeberlen, Jochen Liedtke, Yoonho Park, Lars Reuther, and Volkmar Uhlig. Stub-code performance is becoming important. In *Proceedings of the 1st Conference on Industrial Experiences with Systems Software - Volume 1*, WIESS'00, pages 4–4, Berkeley, CA, USA, 2000. USENIX Association.

[HP86] W. Hwu and Y. N. Patt. Hpsm, a high performance restricted data flow architecture having minimal functionality. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ISCA '86, pages 297–306, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[HPJ03] Yih-Chun Hu, A. Perrig, and D.B. Johnson. Packet leashes: a defense against wormhole attacks in wireless networks. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 3, pages 1976–1986 vol.3, March 2003.

[HW96] Klaus Helwig and Christoph Wandel. High speed content addressable memory. In *Proceedings of the 22nd European Solid-State Circuits Conference*, ESSCIRC'96, pages 300–303, Neuchâtel, Switzerland, Sept 1996.

[IBM64] IBM. *IBM system/360 principles of operation*. IBM Press, 1964.

[IHP] IHP. Das DIAMANT-Projekt. Website. `http://www.diamant-projekt.de/index.html` [Stand: 03.06.2015].

[IHP10] IHP. IQlevel: Innovative high Quality level meter, 2010.

[IHP12] IHP. Aeternitas: Energieeffizientes Wakeup-System für drahtlose Sensorknoten, 2012.

[Inf11] Infineon. *XE166U Derivates*. Infineon Technologies AG, 2011.

[Ins06] Texas Instruments. Msp430x1xx family user's guide, 2006. `http://www.ti.com/litv/pdf/slau049f` [Stand: 18.12.2008].

[Ins11] Texas Instruments. MSP430FR57xx Family User's Guide, May 2011. `http://www.ti.com/lit/ug/slau272c/slau272c.pdf` [Stand: 29.05.2015].

[JM98] Bruce L. Jacob and Trevor N. Mudge. A look at several memory management units, tlb-refill mechanisms, and page table organizations. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 295–306, New York, NY, USA, 1998. ACM.

[JMG+02] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[KAE+14] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, feb 2014.

[KCE92] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. Architecture support for single address space operating systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 175–186, New York, NY, USA, 1992. ACM.

[KCR+10] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental security analysis of a modern automobile. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 447–462, Washington, DC, USA, 2010. IEEE Computer Society.

[Kru15] Andreas Krumholz. *Konzept eines anwendungsspezifisch konfigurierbaren Betriebssystems für drahtlose Sensornetze*. Master thesis, Brandenburgisch Technische Universität Cottbus-Senftenberg, 2015.

[KSW04] Chris Karlof, Naveen Sastry, and David Wagner. Tinysec: A link layer security architecture for wireless sensor networks. In *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 162–175, New York, NY, USA, 2004. ACM.

[KYK+08] Wataru Kanda, Yu Yumura, Yuki Kinebuchi, Kazuo Makijima, and Tatsuo Nakajima. SPUMONE: Lightweight CPU Virtualization Layer for Embedded Systems. In *Proceedings of the 2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing - Volume 01*, pages 144–151, Washington, DC, USA, 2008. IEEE Computer Society.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[Lam71] Butler W. Lampson. Protection. In *Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems*, pages 437–443, New Jersey, NJ, USA, March 1971.

[Law96] Kevin P. Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux Journal*, 1996(29es), September 1996.

[LC02] Philip Levis and David Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95, New York, NY, USA, 2002. ACM.

[Lea10] Neal Leavitt. Researchers fight to keep implanted medical devices safe from hackers. *Computer*, 43(8):11–14, Aug 2010.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[LHSP+09] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, Jochen Streicher, and Olaf Spinczyk. CiAO: An Aspect-oriented Operating-system Family for Resource-constrained Embedded Systems. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 16–16, Berkeley, CA, USA, 2009. USENIX Association.

[Lie93] Jochen Liedtke. Improving IPC by Kernel Design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 175–188, New York, NY, USA, 1993. ACM.

[Lie95a] Jochen Liedtke. Address space sparsity and fine granularity. *SIGOPS Oper. Syst. Rev.*, 29(1):87–90, January 1995.

[Lie95b] Jochen Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

[LMP+04] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. TinyOS: An operating system for sensor networks. In *in Ambient Intelligence*. Springer Verlag, 2004.

[Lop08] Lanfranco Lopriore. Hardware/compiler memory protection in sensor nodes. *International Journal of Communications, Network and System Sciences*, 1(3):235–240, August 2008.

[Lop14] Lanfranco Lopriore. Hardware support for memory protection in sensor nodes. *Microprocessors and Microsystems - Embedded Hardware Design*, 38(3):226–232, 2014.

[LS01] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2001. USENIX Association.

[LS14] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*. 1.5 edition, 2014. `http://LeeSeshia.org`.

[LSH+07] Daniel Lohmann, Jochen Streicher, Wanja Hofer, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Configurable memory protection by aspects. In *Proceedings of the 4th Workshop on Programming Languages and Operating Systems*, PLOS '07, pages 3:1–3:5, New York, NY, USA, 2007. ACM.

[LW01] Jochen Liedtke and Horst Wenske. Lazy Process Switching. In *Proceedings of HotOS-VIII: 8th Workshop on Hot Topics in Operating Systems*, pages 15–18, Elmau/Oberbayern, Germany, May 2001.

[MAK07] René Müller, Gustavo Alonso, and Donald Kossmann. A virtual machine for sensor networks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 145–158, New York, NY, USA, 2007. ACM.

[MBH+05] D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, and O. Rane. IETF_RFC: RFC4226 - HOTP: An HMAC-Based One-Time Password Algorithm, December 2005.

[Men10] Hannes Menzel. *Evaluierung und Implementierung von Konzepten für eine Sensorknoten-spezifische Separierung von Ressourcen*. Master thesis, Brandenburgisch Technische Universität Cottbus, 2010.

[Mil11] Charlie Miller. Battery firmware hacking: Inside the innards of a smart battery. In *Black Hat Briefings*, August 2011. `http://media.blackhat.com/bh-us-11/Miller/BH_US_11_Miller_Battery_Firmware_Public_WP.pdf`.

[MKP07] Misun Moon, Dong Seong Kim, and Jong Sou Park. Computational intelligence and security. chapter Toward Modeling Sensor Node Security Using Task-Role Based Access Control with TinySec, pages 743–749. Springer-Verlag, Berlin, Heidelberg, 2007.

[MMPR11] D. M'Raihi, S. Machani, M. Pei, and J. Rydell. IETF_RFC: RFC6238 - TOTP: Time-based One-Time Password Algorithm, May 2011.

[MRC10] M. Masmano, I. Ripoll, and A. Crespo. An overview of the XtratuM nanokernel. In *Proceedings of the Workshop on Operating System Platforms for Embedded Real-Time Applications*, OSPERT '10, Brussels, Belgium, July 2010.

[MWS04] D.J. Malan, M. Welsh, and M.D. Smith. A public-key infrastructure for key distribution in tinyos based on elliptic curve cryptography. In *Proceedings of the First Annual IEEE Communications Society Conference onSensor and Ad Hoc Communications and Networks*, SECON'04, pages 71–80, Oct 2004.

[NAD+13] 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 *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, pages 479–494, Berkeley, CA, USA, 2013. USENIX Association.

[NCH+05] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, May 2005.

[NDB10] N. Nvaet, B. Delord, and M. Baumeister. Virtualization in automotive embedded systems: an outlook. In *RTS Embedded Systems*, RTS '10, March 2010.

[Ner01] Nergal. Advanced return-into-lib(c) exploits (pax case study). *Phrack Magazin 58*, 4, December 2001.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.

[Nol15] Stephan Nolting. NEO430 Processor (MSP430-compatible). Website, 2015. `http://opencores.org/project,neo430`.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[NZMZ09] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

[One96] Aleph One. Smashing the stack for fun and profit. *Phrack Magazin 49*, 7(14), August 1996.

[OP03] Sejong Oh and Seog Park. Task-role-based Access Control Model. *Information Systems*, 28(6):533–562, September 2003.

[Ora11] Oracle. Sun memory error discovery tool (discovery). Website, 2011. `http://docs.oracle.com/cd/E18659_01/html/821-1784/gentextid-302.html` [Stand 04.08.2015].

[OSM02] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51(1):111–122, Mar 2002.

[PBS+11] Goran Panic, Thomas Basmer, Oliver Schrape, Steffen Peter, Frank Vater, and Klaus Tittelbach-Helmrich. Sensor node processor for security applications. In *Proceedings of 18th IEEE International Conference on Electronics, Circuits and Systems*, ICECS 2011, pages 81–84, Beirut, Lebanon, December 2011.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[PK09] T. Paul and G.S. Kumar. Safe contiki os: Type and memory safety for contiki os. In *Advances in Recent Technologies in Communication and Computing, 2009. ARTCom '09. International Conference on*, pages 169–171, Oct 2009.

[plc14] ARM Holdings plc. Cortex-M1 Processor. Website, 2014. `http://www.arm.com/products/processors/cortex-m/cortex-m1.php`.

[PSB+13] Goran Panic, Oliver Schrape, Thomas Basmer, Frank Vater, and Klaus Tittelbach-Helmrich. TNODE: A low power sensor node processor for secure wireless networks. In *Proceeding of the International Symposium on System on Chip*, ISSoC 2013, pages 1–4, Tampere, Finland, October 2013.

[PSL10] Krzysztof Piotrowski, Anna Sojka, and Peter Langendörfer. Body Area Network for First Responders - a Case Study. In *Proceedings of the 5th International Conference on Body Area Networks*, September 2010.

[PST+02] Adrian Perrig, Robert Szewczyk, J. D. Tygar, Victor Wen, and David E. Culler. Spins: Security protocols for sensor networks. *Wireless Networks*, 8(5):521–534, September 2002.

[PSW+01] Adrian Perrig, Robert Szewczyk, Victor Wen, David Culler, and J. D. Tygar. SPINS: Security Protocols for Sensor Networks. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, MobiCom '01, pages 189–199, New York, NY, USA, 2001. ACM.

[QLZ05] Feng Qin, Shan Lu, and Yuanyuan Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 291–302, Washington, DC, USA, 2005. IEEE Computer Society.

[RCV+05] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. Swift: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 243–254, Washington, DC, USA, 2005. IEEE Computer Society.

[Ren07] Ram Kumar Rengaswamy. *Memory Protection in Resource Constrained Sensor Nodes*. Phd thesis, University of California, 2007.

[RKS07] Ramkumar Rengaswamy, 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, pages 340–349, New York, NY, USA, 2007. ACM.

[RR10] J. Rutkowska and Wojtczuk R. Qubes OS architecture. Technical report, Invisible Things Lab, 2010.

[RRW05] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. *ACM Trans. Embed. Comput. Syst.*, 4(4):751–778, November 2005.

[RSC+07] Ramkumar Rengaswamy, Akhilesh Singhania, Andrew Castner, Eddie Kohler, and Mani Srivastava. A system for coarse grained memory protection in tiny embedded processors. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 218–223, New York, NY, USA, 2007. ACM.

[Rus81] John M. Rushby. Design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 12–21, New York, NY, USA, 1981. ACM.

[Rut08] Joanna Rutkowska. The three approaches to computer security. Website, Sep 2008. `http://blog.invisiblethings.org/2008/09/02/three-approaches-to-computer-security.html`.

[SA03] Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. In *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '02, pages 2–12, London, UK, UK, 2003. Springer-Verlag.

[Sai10] Liu Sainan. Task-role-based access control model and its implementation. In *Proceeding of 2nd International Conference on Education Technology and Computer*, volume 3 of *ICETC'10*, pages V3–293–V3–296, June 2010.

[San93] Ravi S. Sandhu. Lattice-based access control models. *Computer*, 26(11):9–19, November 1993.

[SBPV12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[SBS02] Eugene Shih, Paramvir Bahl, and Michael J. Sinclair. Wake on wireless: An event driven energy saving strategy for battery operated devices. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, MobiCom '02, pages 160–171, New York, NY, USA, 2002. ACM.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29(2):38–47, February 1996.

[SCG+03] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, pages 160–171, New York, NY, USA, 2003. ACM.

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, February 2000.

[Sch14] Marco Schmidt. Cyber-Physical Systems ganz konkret. *elektronikpraxis*, (7), 2014.

[SESS96] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 213–227, New York, NY, USA, 1996. ACM.

[SFK00] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. The NIST Model for Role-based Access Control: Towards a Unified Standard. In *Proceedings of the Fifth ACM Workshop on Role-based Access Control*, RBAC '00, pages 47–63, New York, NY, USA, 2000. ACM.

[SGG12] Oliver Stecklina, Dieter Genschow, and Christian Goltz. TandemStack - A Flexible and Customizable Sensor Node Platform for Low Power Applications. In *Proceedings of the 1st International Conference on Sensor Networks*, Sensornets 2012, Rome, Italy, February 2012.

[Sha07] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Sabrina De Capitani di Vimercati and Paul Syverson, editors, *Proceedings of CCS 2007*, pages 552–61. ACM Press, October 2007.

[SK10] Udo Steinberg and Bernhard Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 209–222, New York, NY, USA, 2010. ACM.

[SKK14] Oliver Stecklina, Stephan Kornemann, and Andreas Krumholz. langOS - A Low Power Application-specific Configurable Operating System. In *Proceedings of the 13. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze*, FGSN'14, Potsdam, Germany, September 2014.

[SKM14] Oliver Stecklina, Stephan Kornemann, and Michael Methfessel. A Secure Wake-up Scheme for Low Power Wireless Sensor Nodes. In *Proceedings of the 4th International Workshop on Mobile Systems and Sensors Networks for Collaboration*, MSSNC'14, Minneapolis, USA, May 2014.

[SLM11] Oliver Stecklina, Peter Langendörfer, and Hannes Menzel. Towards a Secure Address Space Separation for Low Power Sensor Nodes. In *Proceedings of the 1st International Conference on Pervasive and Embedded Computing and Communication Systems*, PECCS'11, Algarve, Portugal, March 2011.

[SLM13] Oliver Stecklina, Peter Langendörfer, and Hannes Menzel. Design of a tailor-made memory protection unit for low power microcontrollers. In *Proceedings of 8th IEEE International Symposium on Industrial Embedded Systems*, SIES'13, pages 225–231, Porto, Portugal, June 2013.

[SLV+15] Oliver Stecklina, Peter Langendörfer, Frank Vater, Thorsten Kranz, and Gregor Leander. Intrinsic Code Attestation by Instruction Chaining for Embedded Devices. In *Proceedings of the 11th International ICST Conference on Security and Privacy in Communication Networks*, SecureComm'15, Dallas, TX, USA, Oct. 2015. Springer.

[SM14] Oliver Stecklina and Michael Methfessel. A Tiny Scale VLIW Processor for Real-time Constrained Embedded Control Tasks. In *Proceedings of the 17th Euromicro Conference on Digital Systems Design*, DSD'14, Verona, Italy, August 2014.

[SMV+11] Oliver Stecklina, Hannes Menzel, Frank Vater, Thomas Basmer, and Erik Bergmann. Hybrid Simulation Environment for Rapid MSP430 System Design Test and Validation using MSPsim and SystemC. In *Proceedings of the 14th*

International Conference on Design and Diagnostics of Electronic Circuits and Systems, DDECS'11, Cottbus, Germany, April 2011.

[SN05] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[Spe05] Bradley Spengler. Increasing performance and granularity in role-based access control systems, 2005.

[SS72] Michael D. Schroeder and Jerome H. Saltzer. A Hardware Architecture for Implementing Protection Rings. *Commun. ACM*, 15(3):157–170, March 1972.

[SS75] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[SSB03] Nik Shaylor, Douglas N. Simon, and William R. Bush. A java virtual machine architecture for very small devices. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, LCTES '03, pages 34–41, San Diego, CA, USA, 2003. ACM.

[SSE+13] Isabella Stilkerich, Michael Strotz, Christoph Erhardt, Martin Hoffmann, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. A jvm for soft-error-prone embedded systems. In *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES'13, pages 21–32, New York, NY, USA, 2013. ACM.

[Ste15a] Oliver Stecklina. A Secure Isolation of Software Activities in Tiny Scale Systems. In *Proceedings of the '8th Annual Ph.D. Forum on Pervasive Computing and Communications' at the '13th IEEE International Conference on Pervasive Computing and Communications'*, PerCom'15, St. Louis, MO, USA, March 2015. IEEE Computer Society.

[Ste15b] Oliver Stecklina. langOS - Low power application-configurable Operating System. Website, 2015. `http://sourceforge.net/projects/langos/` [Stand: 03.06.2015].

[Sti12] Michael Stilkerich. *Memory Protection at Option - Application-Tailored Memory Safety in Safety-Critical Embedded Systems*. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2012.

[STM00] STMicroelectronics. ST16SF48 - Smartcard MCU. Datasheet, 2000. st16sf48a.pdf.

[Tan09] Andrew S. Tanenbaum. *Moderne Betriebssysteme (3. Aufl.)*. Pearson Studium, 2009.

[Ten00] David Tennenhouse. Proactive computing. *Commun. ACM*, 43(5):43–50, May 2000.

[TLKI14] Kuan-Yu Tseng, Dao Lu, Zbigniew Kalbarczyk, and Ravishankar Iyer. AHEMS: Asynchronous Hardware-Enforced Memory Safety. In *Proceedings of the 17th Euromicro Conference on Digital Systems Design*, DSD 2014, Verona, Italy, August 2014.

[TLP05] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press.

[Tog05] Adam Togerson. *Automatic Thread Stack Management for Resource-Constrained Sensor Operating Systems*. Bachelor thesis, University of Colorado, Boulder, 2005.

[TSWSP10] Isabella Thomm, Michael Stilkerich, Christian Wawersich, and Wolfgang Schröder-Preikschat. Keso: an open-source multi-jvm for deeply embedded systems. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 109–119, New York, NY, USA, 2010. ACM.

[Uni85] United States Department of Defense. Trusted Computer System Evaluation Criteria (Orange Book). Technical report, 1985.

[USV+15] R. Urban, M. Schölzel, H.T. Vierhaus, E. Altmann, and H. Selig. Compiler-Centered Microprocessor Design (CoMet) - From C-Code to a VHDL Model of an ASIP. In *Proceedings of 17th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, DDECS '15, Belgrade, Serbia, April 2015.

[Ven00] Vendicator. StackShield, 2000. `http://www.angelfire.com/sk/stackshield` [Stand: 17.09.2015].

[Ven05] Bill Venners. Leading-Edge Java Design Principles from Design Patterns A Conversation with Erich Gamma, Part III. Website, June 2005. `http://www.artima.com/lejava/articles/designprinciples.html`.

[VHM03] R. Venkatasubramanian, J.P. Hayes, and B.T. Murray. Low-cost on-line fault detection using control flow assertions. In *Proceedings of the 9th IEEE Conference on On-Line Testing Symposium*, IOLTS'03, pages 137–143, July 2003.

[vN93] John von Neumann. First draft of a report on the edvac. *IEEE Ann. Hist. Comput.*, 15(4):27–75, October 1993.

[VRSP07] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 273–284, Washington, DC, USA, 2007. IEEE Computer Society.

[WAR06] Yong Wang, G. Attebury, and B. Ramamurthy. A survey of security issues in wireless sensor networks. *Communications Surveys Tutorials, IEEE*, 8(2):2–23, Second 2006.

[Wat01] Robert N. M. Watson. TrustedBSD: Adding Trusted Operating System Features to FreeBSD. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 15–28, Berkeley, CA, USA, 2001. USENIX Association.

[WC08] Nirmal Weerasinghe and Geoff Coulson. Lightweight module isolation for sensor nodes. In *Proceedings of the First Workshop on Virtualization in Mobile Computing*, MobiVirt '08, pages 24–29, New York, NY, USA, 2008. ACM.

[WCA02] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 304–316, New York, NY, USA, 2002. ACM.

[WCC+74] William A. Wulf, Ellis S. Cohen, William M. Corwin, Anita K. Jones, Roy Levin, C. Pierson, and Fred J. Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. *Commununications of the ACM*, 17(6):337–345, 1974.

[WK03] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA*, 2003.

[WKN08] Karsten Walther, Reinhardt Karnapke, and Jörg Nolte. An existing complete house control system based on the REFLEX operating system: Implementation and experiences over a period of 4 years. In *Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation*, ETFA'08, pages 40–45, Sept 2008.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[WRA05] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 31–44, New York, NY, USA, 2005. ACM.

[WS02] Anthony D. Wood and John A. Stankovic. Denial of Service in Sensor Networks. *Computer*, 35(10):54–62, October 2002.

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *In Proceedings of the USENIX Annual Technical Conference*, 2002.

[WSS11] Christian Wawersich, Isabella Stilkerich, and Michael Stilkerich. The Use of Java in the Context of AUTOSAR 4.0. In Katrin Scheinig, editor, *Embedded World Proceedings & Conference Materials*, Nürnberg, Germany, 2011.

[WW05] Richard West and Gary T. Wong. Cuckoo: a Language for Implementing Memory- and Thread-safe System Services. In Hamid R. Arabnia, editor, *PLC*, pages 94–100. CSREA Press, 2005.

[WWC+14] 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 Architecture*, ISCA '14, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press.

[WYX+10] Eric Ke Wang, Yunming Ye, Xiaofei Xu, S. M. Yiu, L. C. K. Hui, and K. P. Chow. Security issues and challenges for cyber physical system. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 733–738, Dec 2010.

[XKPI02] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K. Iyer. Architecture Support for Defending Against Buffer Overflow Attacks. 2002.

[YKC06] Yoshisato Yanagisawa, Kenichi Kourai, and Shigeru Chiba. A Dynamic Aspect-oriented System for OS Kernels. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE'06, pages 69–78, New York, NY, USA, 2006. ACM.

[You96] Charles E. Youman. Rbac transition. In *Proceedings of the First ACM Workshop on Role-based Access Control*, RBAC '95, New York, NY, USA, 1996. ACM.

[YZC08] Yi Yang, Sencun Zhu, and Guohong Cao. Improving sensor network immunity under worm attacks: A software diversity approach. In *Proceedings of the 9th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '08, pages 149–158, New York, NY, USA, 2008. ACM.

[Zah98] A. Zahir. Oil-osek implementation language. In *OSEK/VDX Open Systems in Automotive Networks (Ref. No. 1998/523), IEE Seminar*, pages 8/1–8/3, Nov 1998.

[ZDB09] Vincent J. Zimmer, Shiva R. Dasari, and Sean P. Brogan. Trusted Platforms: UEFI, PI and TCG-based firmware. Technical report, Intel Corporation and IBM Corporation, September 2009.

[ZSJ06] Sencun Zhu, Sanjeev Setia, and Sushil Jajodia. Leap+: Efficient security mechanisms for large-scale distributed sensor networks. *ACM Trans. Sen. Netw.*, 2(4):500–528, November 2006.

APPENDIX A

Applications

A.1 Meetering app

A.1.1 Mapping of software modules onto SAs

Table A.1: Modules of the Meetering app and their usage by software activities

	radio	network	storage	capctrl	main	bootstrap
meetering/meetering						x
meetering/network		x				x
meetering/storage			x			
meetering/capctrl				x		
oslib/main						x
oslib/module	x	x	x		x	x
oslib/dev/cc1101	x					
oslib/hal/clk						x
oslib/hal/fmc			x			x
oslib/hal/gpio	x			x		x
oslib/hal/spi						x
oslib/hal/timer	x	x	x		x	x
oslib/hal/wdt			x			x
oslib/hal/msp430x54x/dco						x
oslib/hal/msp430x54x/nmi					x	
oslib/hal/msp430x54x/fmc			x			x
oslib/hal/msp430x54x/usart	x					
oslib/infra/buffer	x					
oslib/proto/netpkt	x	x				x
oslib/proto/shp		x				x
oslib/proto/csmaca		x				
oslib/svc/alarm		x	x		x	x
oslib/svc/capture					x	x
oslib/svc/pwrmgt	x				x	x
oslib/svc/sched					x	
oslib/svc/time	x	x	x		x	x

A.1.2 Security policy book

Listing A.1: Meetering app security policy book.

```
// activities
user BOOTSTRAP <- oslib::main
user MAIN <- oslib::svc_sched
user RADIO <- oslib::dev_cc1101
user NETWORK <- oslib::proto_csmaca
user CAPTURE <- app::capctrl
user STORAGE <- app::storage

// modules
role B <- {oslib::main, oslib::hal_clk, oslib::hal_msp430x54_dco, \
           oslib::hal_msp430x54x_usart, app::meetering}
role M <- {app::svc_sched, oslib::hal_msp430x54x_nmi}
role N <- {oslib::proto_csmaca}
role C <- {app::capctrl}
role S <- {app::storage}
role R <- {oslib::dev_cc1101, oslib::hal_spi, oslib::infra_buffer, \
                oslib::hal_msp430x54x_usart}
role BM <- {oslib::svc_capture}
role BN <- {oslib::proto_shp, app::network}
role BS <- {oslib::hal_msp430x54x_fmc, oslib::hal_wdt, oslib::hal_fmc}
role BNR <- {oslib::proto_netpkt}
role BMR <- {oslib::svc_pwrmgt}
role BCR <- {oslib::hal_gpio}
role BMNS <- {oslib::svc_alarm}
role BMNSR <- {oslib::svc_time, oslib::hal_timer, oslib::module}

// role assignment
assign RADIO <- {R, BNR, BMR, BCR, BMNSR}
assign BOOTSTRAP <- {B, BM, BN, BS, BNR, BMR, BCR, BMNS, BMNSR}
assign MAIN <- {M, BM, BMR, BMNS, BMNSR}
assign STORAGE <- {S, BS, BMNS, BMNSR}
assign CAPCTRL <- {C, BCR}
assign NETWORK <- {N, BN, BNR, BMNS, BMNSR}

// SA transitions
transition RADIO   <- {BOOTSTRAP, MAIN, CAPCTRL, NETWORK}
transition STORAGE <- {BOOTSTRAP, NETWORK}
transition CAPCTRL <- {BOOTSTRAP, NETWORK, STORAGE, MAIN}
transition MAIN    <- {BOOTSTRAP, NETWORK, STORAGE, RADIO}
transition NETWORK <- {BOOTSTRAP, MAIN}
```

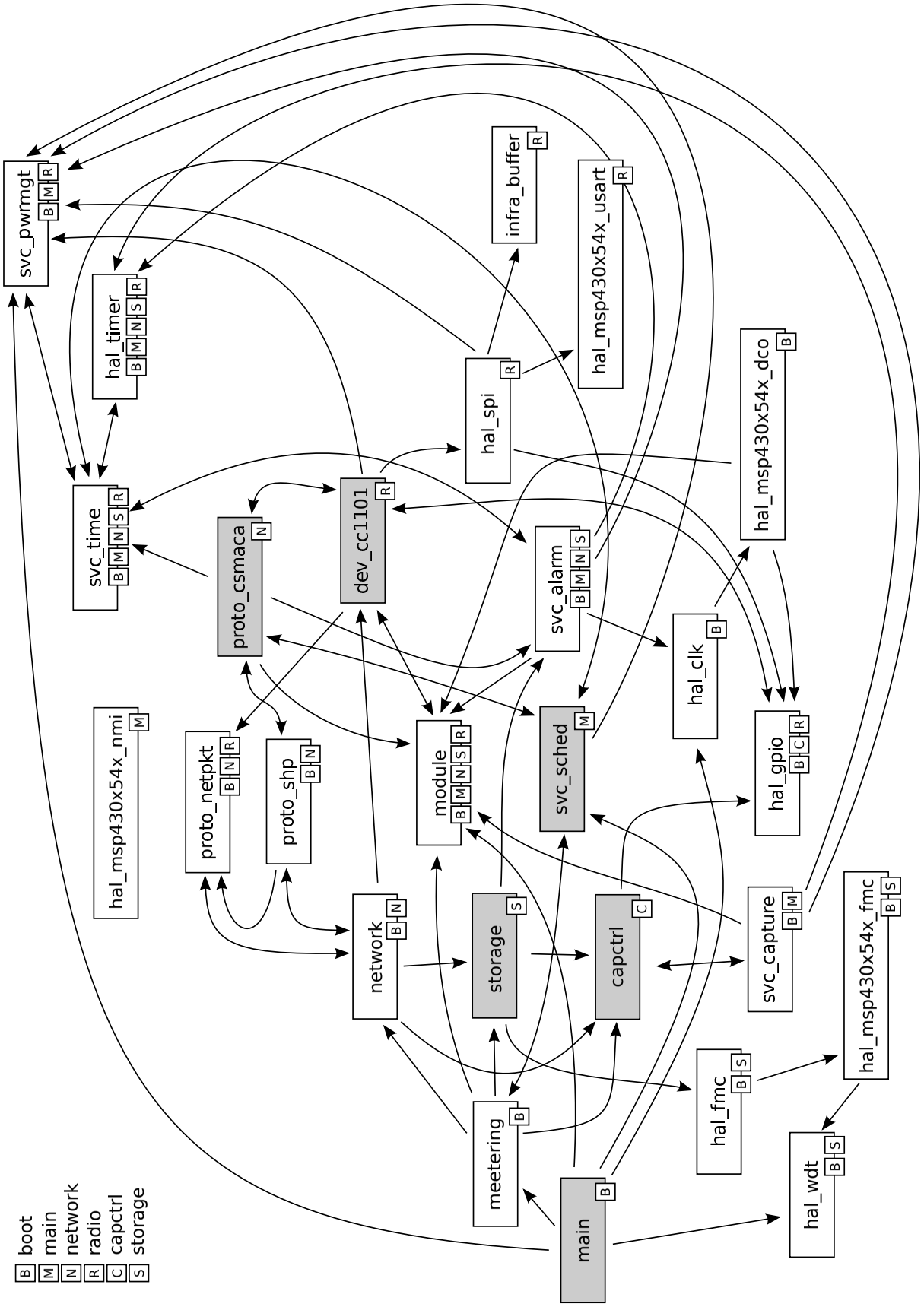

Fig. A.1: Call graph of the Meetering app. The small boxes show the SAs that use the module.

A.2 SWUR firmware

A.2.1 SPD of the SWUR firmware

Listing A.2: SWUR firmware security policy book.

```
user CORE <- core
user TOTP <- totp
user REG <- registers
user SYM <- symdec

role C <- {main, core}
role R <- {registers}
role T <- {totp, hmac, timer, sha1}
role S <- {symdec, spi}
role RS <- {gpio}
role RSTC <- {irq}

allow CORE <- {C, RSTC}
allow TOTP <- {T, RSTC}
allow REG <- {R, RS, RSTC}
allow SYM <- {S, RS, RSTC}

transition CORE <- {TOTP}
transition TOTP <- {CORE, REG}
transition SYM <- {TOTP, CORE, REG}
transition REG <- {CORE}
```

APPENDIX B

langOS interfaces

B.1 langOS Security nucleus

Listing B.1: langOS interface of the security nucleus.

```
/* data space management functions */
typedef unsigned char svc_mpu_ds_t;
typedef unsigned char svc_mpu_perm_t;

int svc_mpudrv_enable(void);
svc_mpu_ds_t svc_mpudrv_ds_get(void *addr);
svc_mpu_ds_t svc_mpudrv_ds_create(uint8_t said, uint32_t base,
                                  unsigned int size);
int svc_mpudrv_ds_append(svc_mpu_ds_t handle, unsigned int offset);
int svc_mpudrv_ds_shrink(svc_mpu_ds_t handle, unsigned int offset,
                         int direc);
int svc_mpudrv_ds_grant(svc_mpu_ds_t handle, svc_mpu_said_t said,
                        svc_mpu_perm_t perm);
int svc_mpudrv_ds_map(svc_mpu_ds_t handle, unsigned char size,
                      svc_mpu_said_t said, svc_mpu_perm_t perm);
int svc_mpudrv_ds_flush(svc_mpu_ds_t handle, uint8_t said);
int svc_mpudrv_ds_delete(svc_mpu_ds_t handle);
int svc_mpudrv_ds_load(svc_mpu_ds_t handle, uint8_t *entry);
```

Listing B.2: MMIO register interface of the MSP430 hardware-based MPU

```
/* MMIO resources */

#define MPUCTRL_              __MSP430_MPU_BASE__ + 0x00
sfrb(MPUCTRL,MPUCTRL_);
#define MPUSAID_              __MSP430_MPU_BASE__ + 0x01
sfrb(MPUSAID,MPUSAID_);
#define MPUADDR_              __MSP430_MPU_BASE__ + 0x02
sfrb(MPUADDR,MPUADDR_);
#define MPUDATA_              __MSP430_MPU_BASE__ + 0x03
sfrb(MPUDATA,MPUDATA_);

#define MPUDDTBR_             __MSP430_MPU_BASE__ + 0x04
sfrw(MPUDDTBR,MPUDDTBR_);
#define MPUSAIDBR_            __MSP430_MPU_BASE__ + 0x06
sfrw(MPUSAIDBR,MPUSAIDBR_);

#define DDT_OP_WRITE     0x00
#define DDT_OP_MAP       0x01
#define DDT_OP_GRANT     0x02
#define DDT_OP_APPEND    0x03
#define DDT_OP_SHRINK    0x04

#define MPUCTRL_EN       0x80
```

Listing B.3: Abstract data structure of an CDC including ACL.

```
typedef enum _svc_mpu_ng_ptype_e {
        svc_mpu_ng_param_in = 0,
        svc_mpu_ng_param_out,
        svc_mpu_ng_param_inout,
        svc_mpu_ng_param_reg
} svc_mpu_ng_ptype_t;

typedef struct _svc_mpu_ng_cdc_s {
        uint8_t param_num;

        void *f_addr;

        struct {
                svc_mpu_ng_ptype_t type;
                uint8_t size;  // svc_mpu_ng_psize_t
        } param[0];
} svc_mpu_ng_cdc_t;

typedef struct _svc_mpu_ng_acl_s {
        uint8_t acl_num;

        struct {
                uint8_t said: 4,
                        role: 4;  // limited to the first 16 roles
        } entry[0];
} svc_mpu_ng_acl_t;
```

B.2 langOS tiny hypevisor

Listing B.4: Data structure of a guarded DDT.

```
typedef struct _guardseg_s guardseg_t;

typedef struct _guardseg_node_t {
    guardseg_t *zero;
    guardseg_t *one;

    unsigned char guardlen;
    unsigned char guard[2];

typedef struct _guardseg_leaf_s {
    unsigned char size;
    unsigned char owner;
    unsigned char said;
    unsigned char perm;
} guardseg_leaf_t;

struct _guardseg_s {
    unsigned char leaf;
    union {
        guardseg_leaf_t l;
        guardseg_node_t n;
    } u;
};
```

APPENDIX C

The tinyVLIW8 MPU

C.1 The DDT look-up engine

Listing C.1: VHDL interface of the DDT of the tinyVLIW8 MPU.

```
entity mpuDdt is
    generic (n: integer := 6);
    port (
        clk : in std_logic;

        memAddr   : in std_logic_vector(11 downto 0);
        memRdEn_n : in std_logic;
        said      : in std_logic_vector (3 downto 0);

        perm  : out std_logic_vector(6 downto 0);
        stall : out std_logic;

        ddtIdx    : in std_logic_vector(n downto 0);
        -- write enable, low active
        ddtWrEn_n : in std_logic;

        -- data bus for writing
        ddtDataOut : out std_logic_vector(30 downto 0);
        -- data bus for reading
        ddtDataIn  : in std_logic_vector(30 downto 0);

        rst_n : in std_logic
    );
end mpuDdt;
```

C.2 tinyVLIW8 timerIRQ app

C.2.1 Waveform

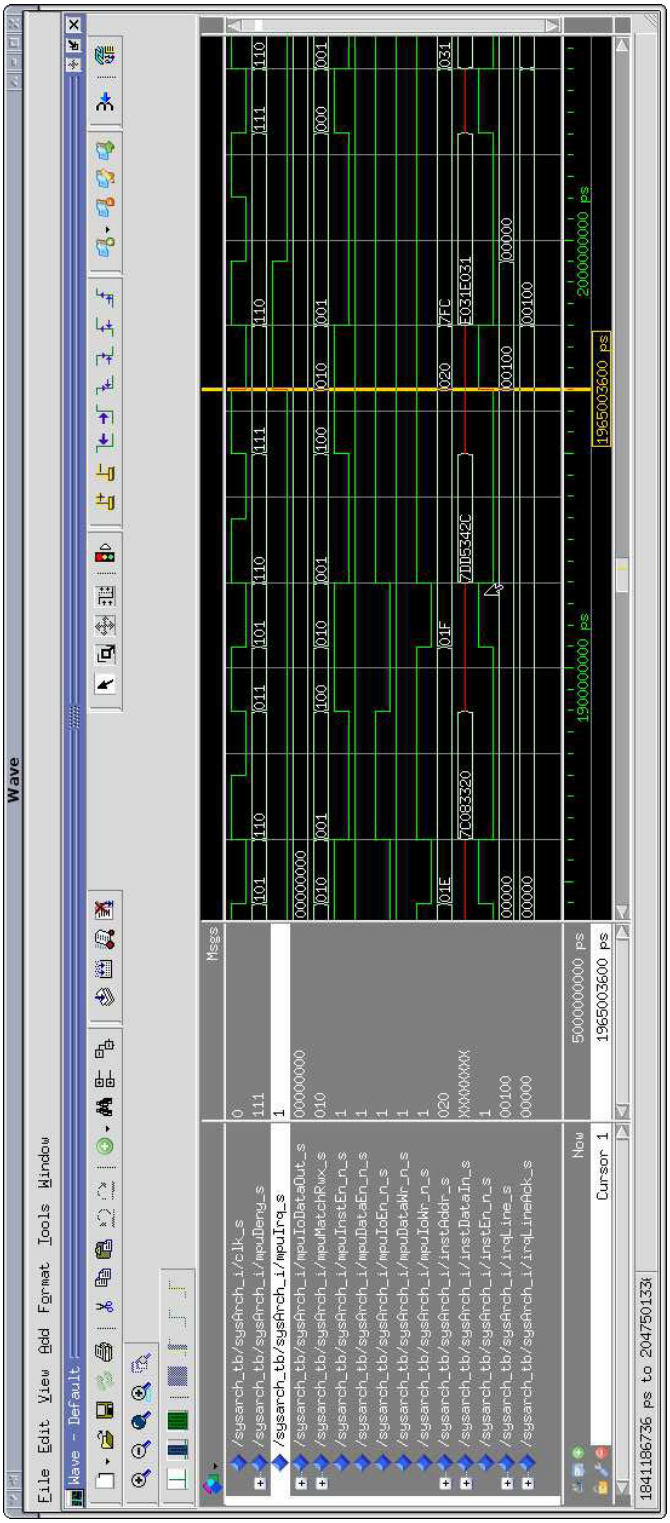

Fig. C.1: ModelSim screen shot of the signal waveform of tinyVLIW8 processor executing an illegal instruction.

C.2.2 Assembler source

Listing C.2: Object dump and assembling code of the timerIRQ example application

```
000:            init:
000: 7b0f7b0f  mov r3, #0x0f;
001: 7e007e00  mov r6, #0x00; // zero register

                   // initialize interrupts
002: 7dc27dc2  mov r5, #0xc2; // configure timer to irq3 and mpu to irq2
003: 35017c08   sti r5, #0x01 | mov r4, #0x08; // enable global interrupt
004: 34003400   sti r4, #0x00;

                   // DDT 0 executable for anybody (base = 0x00, size = 8)
005: 7d007d00  mov r5, #0x00;                  // DDT src idx 0x00
006: 35327c00   sti r5, #0x32 | mov r4, #0x00; // write operation
007: 34337d00   sti r4, #0x33 | mov r5, #0x00; // addr_l
008: 35337c0b   sti r5, #0x33 | mov r4, #0x0b; // addr_h | size
009: 34337d20   sti r4, #0x33 | mov r5, #0x20; // owner | said
00a: 35337c04   sti r5, #0x33 | mov r4, #0x04; // cap(x)
00b: 34333433   sti r4, #0x33;

                   // DDT 1 MPU control read/write access
00c: 7d017d01  mov r5, #0x01;                  // DDT src idx 0x00
00d: 35327c00   sti r5, #0x32 | mov r4, #0x00; // write operation
00e: 34337dc3   sti r4, #0x33 | mov r5, #0xc3; // addr_h
00f: 35337c01   sti r5, #0x33 | mov r4, #0x01; // addr_l | size
010: 34337d20   sti r4, #0x33 | mov r5, #0x20; // owner | said
011: 35337c60   sti r5, #0x33 | mov r4, #0x60; // cap(rw)
012: 34333433   sti r4, #0x33;

                   // DDT 2 allow read/write access to GPIO
013: 7d027d02  mov r5, #0x02;                  // DDT src idx 0x00
014: 35327c00   sti r5, #0x32 | mov r4, #0x00; // write operation
015: 34337dc2   sti r4, #0x33 | mov r5, #0xc2; // addr_h
016: 35337c02   sti r5, #0x33 | mov r4, #0x02; // addr_l | size
017: 34337d21   sti r4, #0x33 | mov r5, #0x21; // owner | said
018: 35337c60   sti r5, #0x33 | mov r4, #0x60; // cap(rw)
019: 34333433   sti r4, #0x33;

                   // set said and enable MPU
01a: 7d017d01  mov r5, #0x01;                  // SAID 0x01
01b: 35317c80   sti r5, #0x31 | mov r4, #0x80; // enable MPU
01c: 34303430   sti r4, #0x30;

01d:            start:            // initialize led (gpio)
01d: 33213321   sti r3, #0x21;                 // P1OUT <= R0
01e: 7c083320  mov r4, #0x08 | sti r3, #0x20; // P1DIR <= R0

                   // initialize and enable timer
01f: 7dd5342c  mov r5, #0xd5 | sti r4, #0x2c; // set CCR0 register
                                                  // clr, cont, ie0, div 4
020: 35283528   sti r5, #0x28;

                   // reset led | initialized counter register r2
021: 33217a40   sti r3, #0x21 | mov r2, #0x40; // P1OUT <= R0
```

```
022:            loop:               // set status register => stall cpu
                                    // ==> addr #0x10   010000b
022: 10001000   ldi r0, #0x00;
023: a880a880   or  r0, #0x80;
024: 30003000   sti r0, #0x00; // ==> sleep

025: 62026202   rla r2, r2; // shift count register
026: f028f028   jc $j0;

027: e022e022   jmp $loop;

028:            j0:                 // toggle led
028: 10211021   ldi r0, #0x21;
029: c801c801   xor r0, #0x01;
                    // ==> addr # 0x18 011000 | reset shift register
02a: 30217a40   sti r0, #0x21 | mov  r2, #0x40;

                // load address of loop in register
02b: e022e022   jmp $loop;

02c:            isr:        // ack timer interrupt
02c: 10291029   ldi r0, #0x29; // load timer ifg
02d: 30293029   sti r0, #0x29; // clr timer ifg

                    // return from interrupt
02e: 36113611   sti r6, #0x11; // clear upper address
02f: 10121012   ldi r0, #0x12; // load return address
030: 30103010   sti r0, #0x10; // restore instruction pointer

031:            nmi:
031: 10301030   ldi r0, #0x30; // read interrupt flags
032: e031e031   jmp $nmi;       // endless loop

                .irq0
7fc: e031e031   jmp $nmi;
                .irq3
7ff: e02ce02c   jmp $isr;
```