KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Secure Resource Sharing for Embedded Protected Module Architectures

Jo Van Bulck

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdspecialisatie Veilige software

Promotor:
Prof. dr. ir. Frank Piessens

Assessoren:
Dr. ir. C. Huygens
Dr. J. Sneyers

Begeleiders:
Ir. J. Noorman
Dr. J.T. Mühlberg

Academiejaar 2015 – 2016

Preface

Before jumping to protected module architectures and secure resource sharing, this is the time to thank some people who have helped me along the way. First off, I would like to thank my advisers Job Noorman and Jan Tobias Mühlberg for their continuous support, guidance, and for all the things they have learned me over the past year. Without our weekly meetings, this thesis would certainly have been less fun. My thanks also goes to my promoter prof. Piessens for allowing me to work on an initially rather vague thesis topic. Furthermore, I would like to thank my family and friends for their encouragements and for listening to my way to long elaborations on the thesis subject. I am also grateful to the reader for taking the time to review my work.

My special thanks goes to the entire open source community for believing in an ideal and for providing the tools without which this thesis would simply be unimaginable. As a final acknowledgement, my thanks goes the following non-exhaustive list of architects for making the world more beautiful and a thesis less severe: Aldo Van Eyck, Álvaro Siza, Bernard Tschumi, Charles & Ray Eames, Charlotte Perriand, Frank Gehry, Gerrit Rietveld, Le Corbusier, Marcel Breuer, Rem Koolhaas, Superstudio, and Walter Gropius.

Jo Van Bulck

Contents

Abstract

Small embedded devices are becoming omnipresent in our daily lives. Through the rise of wireless sensor networks, ubiquitous computing and the Internet of Things, lightweight extensible platforms are increasingly entrusted critical and privacy-sensitive tasks. Yet, to minimise production costs and power consumption, these devices commonly lack hardware support for conventional security measures, such as virtual memory and processor privilege levels.

In this respect, recent research on hardware-level Protected Module Architectures (PMAs) provides an alternative, very lightweight memory protection scheme. These systems allow the execution of security-critical code in protected modules that are isolated from the rest of the system, without relying on a trusted software layer to enforce this separation. While secluding software modules in their own hardware-enforced protection domains allows for strong security guarantees, it also limits their ability to securely share platform resources, such as CPU time or peripheral devices.

This master's thesis explores the feasibility of supplementing the hardware-enforced security guarantees offered by the Sancus PMA with availability and access control guarantees for shared system resources. In contrast to a conventional Operating System (OS), an omnipotent kernel software layer is not introduced. The main contributions of this master's thesis are twofold. First, a generic approach to encapsulate and control access to a shared platform resource is proposed. The approach is implemented and evaluated for a protected file system that can control access to either a shared memory buffer or a shared peripheral flash drive. Second, a secure multithreading model and an accompanying unprivileged scheduler implementation are presented. The scheduler controls access to the CPU time resource by interweaving the execution of logical threads that are conceptually isolated from each other and that might span multiple protection domains.

The work presented in this master's thesis shows that embedded PMAs provide sufficiently strong hardware primitives to not only isolate software modules from each other, but also allow secure implementation of typical OS responsibilities.

Samenvatting

Geïntegreerde computersystemen worden steeds meer alomtegenwoordig in ons dagelijkse leven. Zo worden kleine, uitbreidbare computers steeds vaker ingezet voor kritische en privacygevoelige taken. Echter, omwille van de kostprijs en het stroomverbruik, zijn deze apparaten doorgaans sterk beperkt in hardware-ondersteuning voor gevestigde beveiligingsmaatregelen, zoals virtueel geheugen en beveiligingsringen voor processors.

Recent onderzoek naar geïntegreerde beveiligingsarchitecturen stelt alternatieve lichtgewicht geheugenbescherming voor. Daarbij is het mogelijk kritische code in geïsoleerde, beschermde modules uit te voeren, zonder op software te moeten vertrouwen die deze afzondering afdwingt. Software modules op deze manier in hardware afzonderen laat zeer sterke beveiligingsgaranties toe. Anderzijds biedt het beperkte ondersteuning voor het veilig delen van systeembronnen, bijvoorbeeld processortijd of randapparaten.

Deze masterscriptie onderzoekt de haalbaarheid van het aanvullen van beveiligingsgaranties die worden afgedwongen door de hardware met beschikbaarheids- en toegangscontrolegaranties voor gedeelde systeembronnen. In tegenstelling tot een traditioneel besturingssysteem wordt een almachtige kernel softwarelaag niet geïntroduceerd. De belangrijkste bijdragen van deze thesis zijn tweevoudig. Ten eerste stelt deze thesis een algemene aanpak voor om een systeembron te encapsuleren en de toegang ertoe te controleren. Deze aanpak wordt geïmplementeerd en geëvalueerd voor een beschermd bestandssysteem dat de toegang kan controleren tot ofwel een gedeelde interne geheugenbuffer, ofwel een gedeeld extern flashgeheugen. Ten tweede stelt deze thesis een veilig multitasking model en een bijhorende niet-bevoorrechte scheduler voor. De scheduler controleert toegang tot de processortijd door de uitvoering van logische uitvoeringsthreads te verweven. Deze logische threads zijn conceptueel geïsoleerd van elkaar en kunnen meerdere beschermde modules omvatten.

Het werk gepresenteerd in deze masterproef toont aan dat geïntegreerde beveiligingsarchitecturen voldoende krachtige hardware primitieven bieden om niet alleen software modules van elkaar scheiden, maar ook om veilige besturingssysteemconcepten te implementeren.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

ACL	Access Control List
CFS	Contiki File System
DoS	Denial-of-Service
EA-MPU	Execution-Aware Memory Protection Unit
IoT	Internet of Things
IPC	Inter-Process Communication
ISR	Interrupt Service Routine
MAC	Message Authentication Code
MAL	Memory Access Logic
MMIO	Memory-Mapped I/O
MMU	Memory Management Unit
OS	Operating System
PMA	Protected Module Architecture
SM	Sancus Module
SPM	Self-Protecting Module
TCB	Trusted Computing Base
TCS	Thread Control Structure

Symbols

eIDX	Logical Sancus module entry index
smID	Unique identifier for a Sancus module, assigned by the hardware
SM_{sched}	Sancus module encapsulating the scheduler implementation
SM_{sfs}	Sancus module encapsulating the file system implementation
thrID	Unique identifier for a Sancus thread, assigned by SM_{sched}

Chapter 1

Introduction

Small embedded devices are becoming omnipresent and interconnected in our everyday lives. In an Internet of Things (IoT) where every object is represented by a networked extensible counterpart, adequate software isolation will prove essential to ensure our safety and privacy. Yet, to minimise production costs and power consumption, embedded platforms [17, 20] commonly lack hardware support for established security mechanisms, such as virtual memory and processor privilege levels. Their connectivity and application usages on the other hand make them an interesting target for attackers. It is therefore not surprising that various embedded system security attacks have been described in literature [13, 53, 31, 41].

Due to the lack of lightweight memory isolation techniques, embedded devices generally operate in a *single-address-space* where memory is treated as a global resource, addressable and accessible by everyone. Software running on these devices is thus exposed to modification by malicious or buggy programs. This is especially problematic for reprogrammable devices [21, 10, 15] that feature runtime software extensibility by multiple untrustworthy vendors. As a consequence, recent research on Protected Module Architectures (PMAs) [49, 40, 47, 30] has proposed an alternative memory protection scheme that provides strong, yet inexpensive software isolation for embedded devices. This approach is summarised in Sect. 1.1 and it is explained how current PMAs offer limited support for shared system resources. Section 1.2 thereafter summarises the contributions of this master's thesis that proposes and implements an approach to securely share system resources on embedded PMAs. The structure of the thesis text is outlined in Sect. 1.3.

1.1 Protected Module Architectures

PMAs [49, 47] provide fine-grained memory isolation guarantees in a single-address-space. In short, these architectures allow a programmer to define Self-Protecting Modules (SPMs) with a code and a data section to allow the isolated execution of security-critical code portions. The key characteristic of an SPM is that its private data section is exclusively accessible from its corresponding code section, which can only be entered from a few predefined entry points. An SPM could therefore be

thought of as a standalone enclave in the shared address space. As suggested by its name, a self-protecting module is solely responsible for its own private data section. Attackers from the outside can never directly access the protected data.

PMAs for conventional high-end computer systems have successfully been implemented as an additional layer of protection enforced by a small hypervisor [48] or incorporated in a commodity Operating System (OS) kernel [3]. Hardware-level PMAs [40, 30] on the other hand enforce memory protection through an efficient hardware mechanism [49] that uses the value of the current program counter to decide access to a memory location. As such, hardware-level PMAs can offer an inexpensive, yet substantive memory protection scheme for small embedded devices. Apart from being lightweight, hardware-enforced PMAs are promising because they can isolate SPMs in the shared address space, without relying on any trusted software layer. In this respect, the Trusted Computing Base (TCB) denotes the set of hardware and software components that need to be trusted to ensure the correct execution of a user program. The larger the software part of the TCB, the more likely it contains low-level vulnerabilities that can be exploited using well-known techniques [18] to jeopardise the entire system.

An important advantage of hardware-level PMAs [40, 30] is that they explicitly exclude the OS kernel from the TCB for memory isolation. Confining SPMs in their respective protection domains however also limits their ability to share system resources, such as CPU time or peripheral devices. That is, SPMs should always fulfil their own needs; the only way in which they can get availability or access control guarantees for a platform resource is to claim the resource for themselves. To see how this implies poor flexibility vs. protection guarantees, consider an SPM that denies others access to a peripheral flash drive in order to protect the confidentiality and integrity of its own data, or an SPM that monopolises CPU time to be guaranteed availability at all time. This is evidently undesirable in an embedded context, where system resources are scarce and should be shared among multiple inter-untrustworthy software modules.

1.2 Contributions

This master's thesis explores the possibility of supplementing the hardware-enforced security guarantees offered by embedded PMAs with OS-like availability and access control guarantees for shared system resources. The work of this master's thesis is based upon Sancus [40], a low-cost zero-software PMA explicitly targeted at embedded devices. More specifically, the following contributions are made:

- The minimal set of hardware primitives that need to be provided by a PMA to securely allow the software implementation of OS-like services is identified. An approach to implement OS responsibilities on top of the Sancus architecture is proposed. This approach does not introduce a traditional omnipotent kernel software layer, but rather encapsulates OS concepts in their own unprivileged modules to realise policies not offered by the hardware.

- A generic access control mechanism is proposed to securely share resources that are being accessed through the memory address space. The approach is implemented and evaluated for a protected file system that can control access to either a shared memory buffer, or a peripheral Memory-Mapped I/O (MMIO) flash drive. The security guarantees and general applicability of the resource sharing approach are discussed.
- Sancus' existing implicit control flow model is secured by inserting compiler-generated runtime checks at the boundaries of protected modules.
- A multithreading model for Sancus and an accompanying protected scheduler implementation are provided. The approach allows to control access to the more abstract CPU time resource, supplementing the hardware-enforced security guarantees for SPMs with availability guarantees for logical threads. The security and availability guarantees of the prototype are discussed.

The source code of the protected file system and the secure scheduler is publicly available at `https://github.com/jovanbulck/thesis-src`.

1.3 Outline

The remainder of this text is organised as follows:

Chapter 2: Background This chapter introduces the relevant software security background for this master's thesis. First, conventional software isolation techniques, their downsides and existing mitigations are discussed. Thereafter, PMAs are introduced as an alternative way of isolating software and a general overview of this recent research area is provided.

Chapter 3: Embedded Protected Module Architectures This chapter is dedicated to the problem domain of embedded PMAs and formulates the research objectives. First, a detailed overview of Sancus [40], the development platform for this master's thesis, is provided. Sancus is thereafter briefly compared to Trustlite [30], another hardware-level embedded PMA. The final part of this chapter proposes an approach to securely implement OS-like services on top of these architectures. It is explained how this allows to supplement the hardware-enforced security guarantees for protected modules with software-based availability and access control guarantees for shared system resources.

Chapter 4: Logical File Access Control This chapter presents a protected file system implementation for the Sancus platform. The file system serves as a case study of encapsulating and controlling access to a shared system resource through a lightweight protected software layer on top of hardware-enforced mechanisms. This chapter includes a runtime overhead analysis of the prototype and discuses the security guarantees and general applicability of the resource sharing approach.

Chapter 5: Secure Scheduling This chapter presents a secure multithreading scheme and accompanying scheduler for the Sancus platform. First, existing ideas and challenges to implement multithreading in a protected single-address-space are reviewed. Thereafter, security improvements to Sancus' existing implicit control flow model are described and a logical threading model and scheduler implementation are provided. The final part of this chapter discusses the security/availability guarantees of the current prototype, compares the approach to other PMAs and elaborates on the possibility of future hardware support for preemption.

Chapter 6: Conclusion This chapter concludes the thesis text. The contributions are summarised, limitations are acknowledged and future work directions are provided.

Chapter 2

Background

To enforce security guarantees, a computer system should be able to protect the internal state of a running software entity against other potentially malicious entities. This is referred to as *software isolation*. Traditionally, isolating software is a well-known requirement and an active research field. Consequentially, well-understood solutions for commodity high-end computing platforms have been established over the past decades. The recent rise of low-end embedded devices has triggered research for radically new lightweight software protection mechanisms. This chapter aims at providing an overview of conventional as well as novel software isolation techniques. As such this chapter presents the relevant software security background for this master's thesis.

The explanation is organised as follows. Section 2.1 first discusses established isolation techniques and elaborates on how the associated downsides can be mitigated. Section 2.2 thereafter presents Protected Module Architectures (PMAs) as a novel research field that offers a viable alternative to conventional isolation. Finally, Sect. 2.3 formulates a conclusion for this chapter.

2.1 Conventional Software Isolation

On conventional high-end computing devices software isolation is a fundamental requirement that is typically realised by a trusted OS, backed by advanced hardware support. Section 2.1.1 summarises this traditional approach and Sect. 2.1.2 discusses some of its known drawbacks and alternatives. The previous chapter already mentioned that embedded microcontrollers are constrained by economic and power consumption considerations. They therefore commonly lack the hardware resources needed for traditional software isolation. Section 2.1.3 summarises existing ways of isolating software on these devices.

2.1.1 Operating System Isolation

Conventional computer systems [4, 46, 50] rely on hardware support for processor privilege levels and virtual memory to enforce isolation for the OS and between

different applications. Virtual memory techniques introduce a level of indirection between the (virtual) addresses used by programs and the actual locations in physical memory. A dedicated Memory Management Unit (MMU) hardware component is responsible to automate the virtual-to-physical address mapping. Paged virtual memory, the most widely used set-up, stores the virtual-to-physical translation data in an in-memory page table and supplements the MMU with a translation look-aside buffer to speed up subsequent memory accesses.

Conventional OSs allocate a separate page table per process and update an MMU register that points to the current in-memory page table on context switch. Confining processes in private virtual address spaces has two main advantages. First, each process has the entire virtual address space at its disposal. Processes should therefore not worry about the size and current usage of physical memory. Second, private virtual address spaces provide memory isolation between processes. Indeed, since each process has its own virtual-to-physical address mapping, process A cannot access the physical memory assigned to process B (unless such a mapping is explicitly created to realise shared memory).

Memory protection guarantees via private address spaces of course only hold if user programs are prevented from compromising the OS, or from changing the page table themselves. Conventional computer systems therefore rely on hardware-enforced processor privilege levels, also called protection rings, that allow the OS kernel to run more privileged. Regular user programs can request services from the privileged OS through system calls. Such a system call generates a hardware trap that switches to privileged mode and starts executing the appropriate kernel code.

2.1.2 Issues with Conventional Isolation

The above traditional way of isolating software has several well-known drawbacks and researchers have been trying to mitigate them for decades. The following summarises the major drawbacks and proposed alternatives.

Trusted Computing Base

The above way of isolating software relies on a layered design where user programs trust and rely on the services of the kernel software layer. This implies that a single vulnerability or bug in the omnipotent kernel jeopardises the entire system, as demonstrated by kernel-level malware. In this respect, the TCB denotes the set of hardware and software components that need to be trusted to ensure the safe execution of a software program. The privileged OS kernel software layer thus belongs to the TCB of any user program. This is problematic from a security perspective for two reasons. First, commodity OS kernels are typically written in an unsafe language such as C. This makes them vulnerable to well-known low-level software attacks [18]. Second, commodity OS kernels consist of millions of lines of code[1],

[1]To give an indication of the size and complexity involved: version 3.18 of the Linux kernel consists of almost 19 million lines of code [12] and Microsoft's Windows XP allegedly contains 45 million lines of code [37].

which makes it practically impossible to secure them against these attacks. The large size results from their *monolithic* architecture. That is, most OSs implement all services – scheduling, memory management, file systems, device drivers, etc. – in a single kernel program.

As a response, the *microkernel* architecture [33, 34, 50], depicted in Fig. 2.1, limits the TCB by reducing the size of the trusted kernel layer. The key idea of a microkernel is to implement all non-essential OS services as regular user programs, referred to as *servers*. User programs and servers always communicate indirectly through the microkernel. The privileged microkernel is therefore solely responsible to separate user processes and to provide Inter-Process Communication (IPC) between them. The actual OS services are implemented in user space on top of these abstractions. While microkernels are superior from a security perspective, they have never gained ground in commodity OSs – due to a variety of reasons, including complexity and initial performance issues [34].

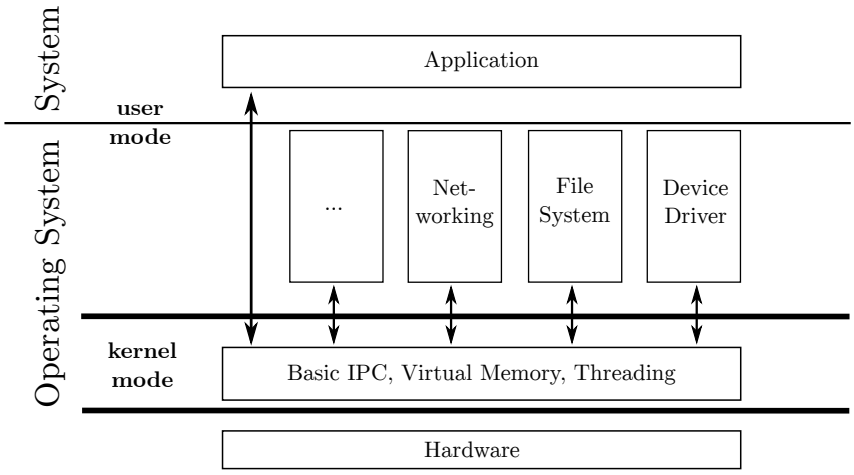

FIGURE 2.1: Microkernel architecture with server processes in user space

The TCB consequences of a layered monolithic architecture can also be mitigated by enforcing a modular design. Witchel et al. [54] employ a combination of hardware and software techniques, known as Mondriaan memory protection, to enforce memory isolation of Linux kernel modules. Their approach prohibits a kernel module from accessing another module's memory directly. A vulnerability in one such module (e.g. a device driver) does therefore not jeopardise the entire monolithic kernel program.

Apart from reducing the kernel's size, another strategy to increase its trustworthiness is by formally verifying parts of it. Research on formal verification [43, 38] has indeed proven memory safety of parts of the Linux kernel. While completely verifying a large monolithic kernel such as Linux is infeasible, the functional correctness of the seL4 microkernel implementation has been entirely verified [28]. From a security perspective, such a verified OS kernel remains part of the TCB, but becomes trustworthy. That is, applications that build upon the kernel are provided with strong formalised assurances, as opposed to the implicit entangled TCB of legacy OSs.

Coarse-Grained Protection

In a conventional OS that provides software isolation via private address spaces, the process as the unit of multiprogramming coincides with the unit of memory protection. In the context of increasingly complex, extensible and object oriented applications however, one could argue that this unit is too coarse-grained. A 1994 paper already addresses the problem: "private-address-space systems force poor tradeoffs between protection, performance, and integration. [...] applications need better control of protection and sharing than current systems can provide." [9].

A private virtual address space per process is needed to be able to re-use virtual addresses in restricted address spaces ($\leq$ 32 bit). The rise of wide-address (64 bit) architectures in the early nineties however rendered this use of private address spaces superfluous. As a result, researchers [8, 9] devised single-address-space OSs that place all processes in a single global virtual address space and realise alternative fine-grained protection domains on top. All programs execute in the same address space, but their memory access rights within this shared address space are determined by their current protection domain. As such, single-address-space OSs separate addressability from accessibility.

Capability-based systems represent another approach to provide fine-grained memory access control in a shared address space. While capability-based systems are an historical [7] research direction, they are currently reviving. The recent CHERI [55] system features a hardware-based capability implementation that offers fine-grained protection domains inside the virtual address space of a conventional OS process. Access rights for memory blocks are represented by *capabilities* and an executing program is granted access to all memory that is described by its current set of capabilities. A capability can be thought of as an unforgeable memory pointer that is specially tagged and that includes additional meta data, such as an offset and permission fields. Capabilities describe access rights to a memory block and are themselves stored in memory. A program that holds a capability is therefore granted access to the corresponding memory block, which may contain other capabilities to extend the memory access rights once more. To prevent applications from defining their own memory access rights, valid capabilities include a special tag that can only be set by a privileged OS kernel [7] or through special hardware instructions [55]. Since capabilities can be passed freely along, or used to create more restrictive capabilities, capability-based addressing simplifies sharing, but makes revocation of previously assigned access rights more difficult.

2.1.3 Embedded Software Isolation

Due the limited resources and specialised needs of embedded devices, a heterogeneous range of small dedicated OSs [20, 56, 17] has emerged in recent years. Initially, these OSs assumed a single, static application. Hence, no need for memory protection: the embedded application was statically linked with the OS into a single monolithic image, sharing an unprotected single-address-space [20]. While embedded OSs nowadays commonly feature concurrency and dynamic reprogramming to update or install

new applications at run time [21, 10, 15], support for memory protection is non-existing or remains very limited. The reason is that resource-constrained embedded microcontrollers generally lack hardware support for established security mechanisms, such as virtual memory and CPU privilege levels. Several authors [17, 21, 24] foretell that these hardware constraints will endure, due to low-power requirements.

Considerable research effort has been put in providing software isolation and OS protection for embedded microcontrollers that lack related hardware support. Safe TinyOS [11] provides efficient type and memory safety by modifying code at compile time. t-Kernel [24] on the other hand provides virtual memory, preemptive scheduling and OS protection by modifying untrusted application code at load time. Both approaches decrease performance and rely on a software TCB to enforce memory protection. The next section introduces a novel hardware memory protection mechanism suitable for efficient runtime memory access control on small devices.

2.2 Protected Module Architectures

The above discussion introduced several issues concerning the way software isolation is traditionally realised. Consequently, recent research on Protected Module Architectures (PMAs) [49, 47] attempts to realise fine-grained memory protection guarantees with a moderate-sized TCB. In short, these architectures allow a programmer to define standalone modules with a code and data section so that security-critical code fragments can be executed in their own protection domain, isolated from the rest of the system.

The explanation is organised as follows. Section 2.2.1 first introduces the concept of a protected software module. Section 2.2.2 thereafter elaborates on the provided security guarantees and Sect. 2.2.3 finally provides an overview of the existing PMA implementations.

2.2.1 Self-Protecting Modules

A Self-Protecting Module (SPM) corresponds to a fine-grained protection domain in a shared address space. The layout of an SPM is depicted in Fig. 2.2 and consists of two contiguous memory sections: a public *text* or *code section* containing a fixed number of *entry points* and a private *data section*. The key characteristic is that an SPM's private data section is exclusively managed by its corresponding code section, which is only entered from a few predefined entry points. An SPM could therefore be thought of as a standalone enclave in unprotected memory. Attackers from outside can never directly access the protected data; a self-protecting module is solely responsible for its own private data section, hence its name.

The concept of SPMs is of course only useful if it can be enforced in a reliable and efficient way. In this respect, program counter based access control [49] is a novel lightweight memory protection technique that uses the current value of the program counter to decide access to a memory location. An SPM can be unambiguously defined in the single-address-space through simple program counter based access control rules in an access control matrix. Table 2.1 lists such memory access rights

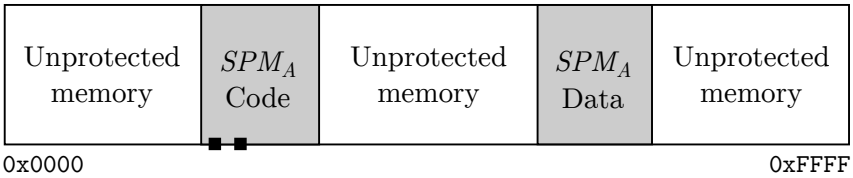

Figure 2.2: Layout of a protected module SPM_A in the shared address space (gray areas represent protected memory and black squares represent entry points)

for an SPM in the traditional UNIX notation. All memory addresses are categorised as either "protected", that is belonging to the SPM, or "unprotected". Protected memory is further subdivided as belonging to an entry point, a code section or a data section. The columns of the table categorise the memory location being accessed, whereas the rows categorise the current value of the program counter. Note that when deciding access to an SPM the program counter is only categorised as "protected" when corresponding to an address within the code section of the *same* module (i.e. any other module is treated as if it where unprotected). The program counter based rules in Table 2.1 thus enforce that (*i*) the code section of an SPM is read-only, (*ii*) program flow only enters an SPM through one of its predefined entry points, and (*iii*) the data section of an SPM is only accessible when executing in its corresponding code section.

Table 2.1: Program counter based access control rules for an SPM

From \ to	Protected			Unprotected
	Entry	Code	Data	
Protected	`r-x`	`r-x`	`rw-`	`rwx`
Unprotected / other SPM	`r-x`	`r--`	`---`	`rwx`

2.2.2 Security Guarantees

SPMs are realised by a Protected Module Architecture (PMA) implementation, as discussed in the next section. Such a PMA typically provides SPMs with the following security guarantees.

Secure Control Flow As explained above, the code section of an SPM can only be entered via a few predefined entry points. That is, an attacker cannot jump to arbitrary code in the SPM. This prevents him from abusing useful code snippets [45] that operate on private data or from bypassing security-sensitive code (such as access control or encryption functionality). Furthermore, SPMs should maintain their own private call stack to save i.a. function return addresses. Such a safe stack, as initially proposed by Kumar et al. [32] and formalised by Agten et al. [2], ensures that control flow within an SPM protection domain cannot be influenced from outside.

Secrecy of Private Data SPMs are provided with exclusive read and write access to their private data section. This allows a programmer to implement for example a cryptographic module that saves the secret key in its private data section and safely offers encryption/decryption facilities towards others. Such an SPM remains responsible however for the correct implementation of its code section. More specifically, to ensure confidentiality and integrity of the private data section, the corresponding code section should not leak private data and should not be vulnerable to low-level security attacks [18] such as buffer overflows that might tamper with the private data or alter control flow within an SPM. Consequently, the TCB of an SPM consists of (*i*) the TCB needed to realise the PMA implementation that enforces the SPM's access rights, and (*ii*) the SPM's own code section.

Authentication To build flexible trustworthy systems, reliable authentication of running SPMs is essential. To do so, support from the PMA implementation is needed. The exact authentication mechanism and guarantees therefore depend on the specific PMA [49, 40, 48, 30]. In general however, to unambiguously identify a running SPM, one minimally needs (*i*) a cryptographic hash of the code section, and (*ii*) the exact load addresses of the code and data sections (since the content of the relocatable code section depends on it). Once provided with this information, a cryptographic hash of the publicly readable code section can be calculated and compared to authenticate the identity of the running SPM. For reliable authentication however, one still needs a guarantee that the SPM being verified is still loaded and protected correctly. Such a guarantee should be provided by the supporting PMA's interface, as explained further on.

How exactly the above information is provided depends on the PMA implementation. Strackx et al. [49, 48] for example propose to accompany each SPM with a security report that contains i.a. the memory layout and a cryptographic hash of the code section. Since the security report is signed with the private key of the issuer, SPMs can verify the trustworthiness of the report through a chain of trust to a known certificate authority. Noorman et al. [40] avoid the use of public key cryptography by deploying the calling SPM with a Message Authentication Code (MAC) of the code section and load addresses of the callee.

Secure Communication Strackx et al. [49] show how to set up a secure local communication channel between SPMs that ensures mutual authentication, confidentiality and integrity of the passed data. Their approach relies on reliable SPM authentication and assumes the absence of interrupts. A one-way authenticated channel is set up as follows. First, the calling SPM authenticates the SPM being called. Next, the caller jumps to the callee's desired entry point, passing any arguments safely through CPU registers. Finally, the callee returns to the caller. To do so however, the callee needs the return address. Recall that SPMs have their own private call stack and can only be entered from a few predefined entry points. This implies that (*i*) the calling SPM should pass the return address as an argument, and (*ii*) this continuation point cannot simply be the address following the call instruction. Strackx et al. [49] therefore propose to pass the address of a special

return entry point in the calling SPM that restores the internal private call stack and continues internal execution after the call instruction. Moreover, the callee can use the provided address of this entry point to identify and verify the calling SPM, resulting in a mutually authenticated communication channel for return values.

Note that confidentiality and integrity of the passed data is guaranteed because the caller places them in CPU registers before executing an unconditional jump to the callee. This implies that – in the absence of interrupt – only the caller and the callee can see or modify the data. Passing data via CPU registers of course limits the number and size of the arguments. To overcome these limitations, multiple subsequent calls may be used or data can be passed through unprotected memory in encrypted form.

2.2.3 Implementations

As mentioned above, the security guarantees for SPMs should be implemented by a Protected Module Architecture (PMA). Such a PMA can be realised on different levels, depending on the application scenario. This section discusses PMA implementations in hardware, through a trusted hypervisor and incorporated in a trusted OS kernel. Each of these implementations has its own advantages in terms of performance, cost and portability. Moreover, since the PMA enforces the access control rights for SPMs, it is naturally incorporated in the TCB. The choice of the implementation level therefore also defines the size of the TCB.

Hardware-Level Implementation

Hardware-level PMAs implement some form of program counter based access control in hardware and extend the instruction set to allow the safe creation of SPMs in the shared address space. The prime advantage of these PMAs is that they feature a small hardware-only TCB. SPMs in these systems indeed rely solely on the correct functioning of the hardware for their security guarantees. Moreover, the modified memory access semantics can be enforced efficiently through a lightweight program counter based access control hardware mechanism that does not increase the memory access time [40, 30]. One of the downsides of a custom hardware implementation however, is that portability of legacy applications is obviously hindered.

Recall from Sect. 2.1.3 that embedded platforms commonly lack hardware support for virtual memory and therefore need an alternative low-cost software isolation scheme. In this context, hardware-only PMAs look particularly interesting as they provide a lightweight, yet substantive memory protection mechanism. Strackx et al. [49] first proposed the general idea of SPMs and program counter based access control as an efficient memory isolation technique for embedded devices. Noorman et al. [40] presented Sancus, a hardware-only PMA explicitly targeted at small embedded devices and the development platform for this master's thesis. Koeberl et al. [30] presented Trustlite, a hardware-enforced PMA that features an execution-aware memory protection unit for small computing devices. Given the importance of the

embedded PMA research field for this master's thesis, a detailed explanation of the Sancus platform and a comparison to Trustlite are presented in Chapter 3.

The recently proposed Intel Software Guard Extensions (SGX) [36, 27] represent a different research direction that employs a hardware-level PMA. In contrast to the embedded approach above, SGX is a set of hardware extensions for high-end multiple-address-space PCs and servers. SGX does not consider program counter based access control as a standalone memory protection mechanism, but rather as a means to complement the existing coarse-grained virtual memory protection scheme. SGX thus allows for hardware-enforced security guarantees in a conventional high-end untrusted execution environment. To this end, SGX extends the Intel architecture with new instructions that allow the creation, destruction, entering and exiting of hardware-protected enclaves within an application's private virtual address space. SGX's hardware model allows a conventional OS to take care of virtual memory translation and swapping of enclave pages, but regards such an OS as an untrusted agent. Furthermore, SGX enclaves feature multiple internal control flow threads that can be interrupted at any time, as discussed in more detail in Sect. 5.1.3.

Hypervisor-Level Implementation

The Fides system [48] represents another way of implementing PMAs through a small hypervisor software layer that isolates SPMs in a separate secure virtual machine. Such a hypervisor-level implementation has the advantage that it does not require changing the hardware, at the cost of only a moderate-sized TCB. Fides allows a programmer to define seamlessly integrated SPMs within an application's virtual address space. Fides therefore demonstrates the feasibility of realising PMAs while remaining compatible with legacy hardware and OSs.

The Fides architecture consists of an omnipotent hypervisor bottom layer that keeps track of two separate virtual machines, one for the legacy OS and one for a security kernel that manages SPMs. Both virtual machines have the same view on physical memory, but the hypervisor enforces different access rights. It makes sure that the memory belonging to an SPM cannot be accessed by the legacy OS. The hypervisor thus enforces coarse-grained memory protection guarantees, whereas the fine-grained SPM access control model is enforced by a small dedicated security kernel. As such, secure SPM execution relies on the hypervisor implementation as well as the security kernel, which together represent a relatively small TCB of 7,159 lines of code [48].

The key advantages of the Fides architecture are its compatibility with legacy hardware and software and the fact that the legacy OS is excluded from the TCB for secure SPM execution. Moreover, the performance penalty is acceptable since the hypervisor only needs to switch the virtual machines when entering or exiting an SPM. The Fides architecture inevitably depends on hardware support for processor privilege levels and virtual memory however. This makes a hypervisor-level solution unsuitable for embedded devices.

Kernel-Level Implementation

The Salus system [3] provides a PMA implementation by extending the Linux kernel. Salus employs SPMs as an additional fine-grained layer of protection inside the virtual private address space of a Linux process. Recall from Sect. 2.1.2 that an application programmer in a conventional multiple-address-space OS environment is obligated to compromise between protection and performance. The key advantage of Salus is therefore that it allows a single process to consist of multiple logical protection domains where a vulnerability in one SPM cannot affect the others. To this end, Salus also includes a privilege containment mechanism to limit the system calls that can be used by a specific SPM.

While such an implementation is interesting from a pragmatic point of view, it also enlarges the TCB. In contrast to the above PMAs, Salus does not execute SPMs in an isolated part of the system. A single vulnerability in the omnipotent monolithic kernel layer indeed suffices to invalidate the security guarantees for SPMs. Moreover, a kernel-level implementation inevitably relies on hardware-support for processor privilege levels and is therefore unsuitable for embedded devices.

2.3 Conclusion

Adequate software isolation through memory protection is imperative to enforce security guarantees for a shared computing platform. Well-understood protection mechanisms have been established for classical computer systems. The most widespread approach isolates processes in their own private virtual address space through a trusted OS that is shielded from the applications. Important well-known disadvantages of such an approach include (*i*) a large TCB that incorporates the OS kernel, and (*ii*) the lack of additional fine-grained protection domains inside the virtual address space of a process. Moreover, the approach inevitably relies on hardware support, which is not widely available for embedded systems – to minimise production costs, as well as power consumption.

Consequentially, recent research on PMAs attempts to provide inexpensive and fine-grained protection domains in a shared address space, while keeping the TCB small. This chapter discussed PMAs that allow hardening security critical applications on conventional high-end computers, but also mentioned lightweight hardware-level PMAs that can offer a standalone substantive memory protection model for low-end embedded devices. The next chapter elaborates further on embedded hardware-level PMAs and the consequences of a zero-software TCB that excludes the OS kernel.

Chapter 3

Embedded Protected Module Architectures

The previous chapter introduced the general research area on PMAs and their relevance in an embedded context. Since this master's thesis builds upon Sancus, a hardware-enforced PMA for small embedded devices, this chapter is entirely dedicated to the problem domain of embedded hardware-level PMAs and formulates the research objectives.

The chapter is organised as follows. Section 3.1 first provides a detailed description of Sancus, the development platform for this master's thesis. Next, Sect. 3.2 briefly contrasts the Sancus architecture with Trustlite, a similar embedded hardware-level PMA with a different access control approach. Section 3.3 thereafter discusses the consequences of a zero-software TCB that excludes the OS kernel and explains how this requires reconsidering traditional OS concepts. This section thus contextualises the goal for this master's thesis: providing secure resource sharing for embedded PMAs without introducing an omnipotent trusted kernel software layer. Section 3.4 finally concludes.

3.1 Sancus

The previous chapter already mentioned the Sancus platform [40] as a hardware-only PMA implementation that is explicitly targeted at low-end extensible devices. Sancus extends the memory access model and instruction set of a TI MSP430 microcontroller to provide hardware-enforced memory isolation guarantees for Sancus Modules (SMs). Note that an SM is a realisation of the abstract concept of a Self-Protecting Module (SPM), as introduced in Sect. 2.2.1. Therefore, everything presented here for SMs can also be mapped on the more generic concept of SPMs.

The work presented in this master's thesis is built upon Sancus for two major reasons. First, Sancus is an active research project at KU Leuven and its source code is publicly available [39], allowing for extensibility where needed. Second, Sancus features unique caller authentication hardware instructions that will prove essential for this master's thesis.

The following provides a detailed overview of the Sancus system and its internals.

3.1.1 Attacker Model

Sancus protects the internal state of SMs against a powerful attacker which may deploy or tamper with any software running on the node. More specifically, an attacker can address the whole single-address-space, but Sancus' modified memory access logic may disallow access to protected memory depending on the current value of the program counter. An attacker may control all unprotected code, for example the node's OS or shared libraries such as `libc`. Since SMs do not necessarily have to trust each other, Sancus also protects against an attacker that can deploy arbitrary malicious SMs on the node. Finally, an attacker may control the communication channel between the node and the software provider.

Sancus does not protect against hardware-level attacks however. An attacker with physical access to the node may for example extract memory content via a cold-boot attack [25]. Sancus, like other PMAs, does also not protect SMs against implementation vulnerabilities in their own code section, as previously discussed in Sect. 2.2.2.

3.1.2 Dynamic Deployment Model and Key Management

Sancus is especially interesting for small devices that feature extensibility with software modules from multiple inter-untrustworthy stakeholders. In such a system, adequate isolation of SMs in the shared address space is indeed essential. Sancus supports a generic multiple stakeholder model where an infrastructure provider IP governs a number of low-end computing devices, referred to as nodes N_i. External software providers SP_j that are recognised by IP may deploy software modules $SM_{j,k}$ on the nodes.

Sancus allows a secure mutually authenticated communication channel between SMs running on the same node and between an SM and its remote software provider. To this end, Sancus features hardware extensions and a key derivation scheme to establish a symmetric cryptographic key that is shared by an SM and its software provider SP. The infrastructure provide IP acts as a trusted party that shares a symmetric key K_N with each of its administered nodes. To make sure the node's key K_N is kept secret at all time, K_N is exclusively managed by hardware on the Sancus-enabled node. IP uses its copy of K_N and a non-secret key derivation function *kdf* to generate a new key for each software provider, identified by a non-secret SP identifier:

$$K_{N,SP} = kdf(K_N, SP) \tag{3.1}$$

IP distributes these keys to the corresponding software providers. In the end, the software provides want to share a symmetric key with the SMs they deploy on IP's nodes. To so, they need to know the module's *identity*, consisting of (*i*) the module's code section, and (*ii*) the load addresses of the code and data sections. An SM's identity may not be known until after the module is loaded in a dynamic deployment

scenario. In such a case, untrusted software running on the node may simply send a symbol table containing the load addresses to the software provider. Provided with $SM_{identity}$ and $K_{N,SP}$, the software provider generates a module-specific key:

$$K_{N,SP,SM} = kdf(K_{N,SP}, SM_{identity}) \tag{3.2}$$

It follows from Eqs. (3.1) and (3.2) that provided with the non-secret SP and $SM_{identity}$ information and the secret K_N key, a hardware implementation of the kdf function can calculate the module-specific key on the node as follows:

$$K_{N,SP,SM} = kdf(kdf(K_N, SP), SM_{identity}) \tag{3.3}$$

Sancus-enabled nodes stores one such key for each SM in a protected storage area that is only indirectly accessible through hardware instructions. Since $K_{N,SP,SM}$ depends on the identity of the corresponding SM, the load process should not be trusted. That is, if the load process changes the content of the code section before enabling protection or sends false load addresses to the software provider, the node's hardware and the software provider won't share a symmetric key. This follows from the fact that a software provider knows the content of the code section of the SM it deploys and combines this with the load addresses to generate his key $K_{N,SP,SM}$ according to Eq. (3.2). This key will differ from the hardware-generated key $K_{N,SP,SM}$ of Eq. (3.3) if either the content of the code section or the load addresses differ. As explained further on, the module-specific key $K_{N,SP,SM}$ can therefore safely be used for local or remote authentication.

3.1.3 Sancus Module Isolation

As discussed in Sect. 2.2.2, PMAs offer two basic security guarantees towards SMs: entry point restriction and secrecy of the private data section. Sancus implements entry point restriction in hardware by enforcing program flow can only enter an SM through the start address of its corresponding code section, referred to as the *physical* entry point. Section 3.1.7 explains how SMs employ this single physical entry point for private call stack switching and multiplexing of multiple *logical* entry points. Sancus furthermore implements program counter based memory isolation by means of a custom Memory Access Logic (MAL) hardware circuit for every SM. These circuits use simple combinational logic to ensure that the private data section of an SM is only accessible when the current value of the program counter lies within the bounds of its corresponding code section. The layout information of an SM (i.e. start and end addresses of the code and data sections) is stored in dedicated registers of a protected storage area that is only accessible from hardware. The use of parallel combinational hardware circuits ensures that the modified memory access semantics are enforced without increasing memory access time. The maximum number of SMs is imposed at hardware synthesisation time, since Sancus needs one MAL circuit per SM.

After loading an SM's code section in memory, a programmer can use the `sancus_enable` hardware instruction to enable memory protection for the SM. This

instruction requires the SM's layout as an argument, i.e. the start and end addresses of the module's code and data sections. After verifying that the provided address ranges do not overlap with existing SMs, the instruction stores them in the protected storage area to instantiate a new MAL circuit. The `sancus_enable` instruction furthermore requires the software provider's SP identifier as an argument so that it can generate the module-specific key $K_{N,SP,SM}$ according to Eq. (3.3). This key is stored in the protected storage are as well and may be used by other hardware instructions. Recall from the above that the hardware-generated key will differ from the one of the software provider if an attacker tampers with the module before enabling protection. Finally, the `sancus_enable` instruction zeroes out the content of the private data section to make sure an attacker does not have any influence on the initial state of a module. The `sancus_enable` instruction returns a status flag to indicate whether the call was successful. Figure 3.1 provides a graphical representation of the protected storage area after enabling a module SM_1.

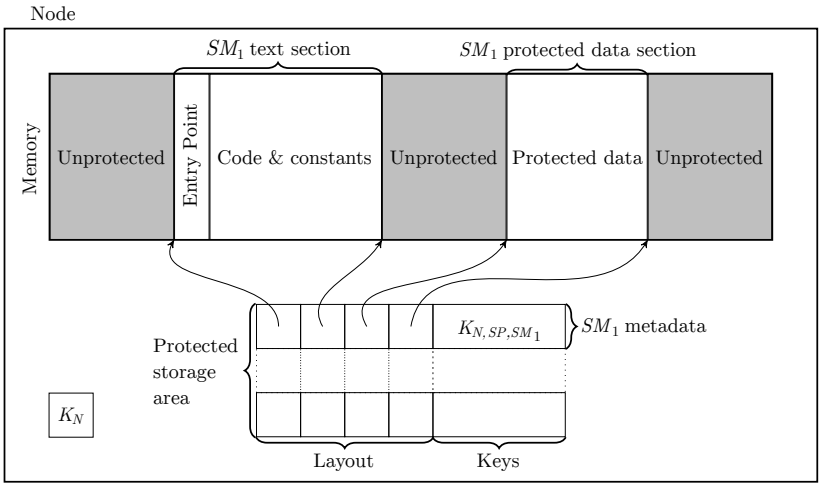

Figure 3.1: Layout of a Sancus module SM_1 in the shared address space and protected storage area (from [40])

Since SMs can choose their own layout (on a first-come, first-served basis), Sancus' generic memory protection scheme can be used to secure Memory-Mapped I/O (MMIO) peripherals that are being accessed through the memory address space. It suffices indeed to include the relevant MMIO addresses in the private data section of an SM to provide it with exclusive access to the device. There is one pitfall however: the MAL circuits presented above currently only allow a single contiguous private data section per SM. This implies that a module including a MMIO address range in its private data section, cannot at the same time store protected data. Moreover, as it cannot safely provide the call stack needed by higher level programming languages, its corresponding code section should be entirely implemented in assembly. These limitations call for small dedicated driver SMs that restrict access to MMIO peripherals, as illustrated in the next chapter.

A programmer can use the `sancus_disable` instruction to disable memory pro-

tection for the currently executing SM, if any. This instruction clears the corresponding entry in the protected storage area and disables the MAL circuit. An SM should clear any remaining secrets in its private data section itself before calling the `sancus_disable` instruction.

When one of the MAL circuits detect a memory access violation, a non-maskable interrupt is generated.[1] On such an interrupt, the hardware jumps to the address of the Interrupt Service Routine (ISR) that is registered in the corresponding entry of the interrupt vector table (which is at a fixed location in the TI MSP430's memory). Since the currently executing program performed an illegal memory access, it cannot be continued normally and the ISR typically halts the system. Remark that Sancus currently offers no availability guarantees whatsoever, an attacker or buggy SM can for example perform an illegal memory access to halt the system or execute an infinite loop to monopolise CPU time.

3.1.4 Secure Linking

Sancus features a strong authentication mechanism that allows a module SM_1 to reliably call another module SM_2 running on the same node. More specifically, SM_1 can be ensured before calling SM_2 that (*i*) protection has been correctly set up for SM_2 as explained above, and (*ii*) the code section of SM_2 has not been tampered with. To this end, Sancus offers a `sancus_verify_address` instruction that requires two arguments: the expected address of SM_2 and a MAC of the identity of SM_2 calculated with the module-specific key of SM_1. Since the software provider of SM_1 knows its module-specific key, it can supply SM_1 with the MACs of all the SMs it needs to call, either at deployment time or at run time (to be explained below). The `sancus_verify_address` instruction first verifies whether an SM is currently loaded at the expected address. If this is the case, the instruction computes the MAC of that module using the module-specific key of the calling SM. Finally, the instruction compares the provided MAC with the computed MAC and returns zero if they are not equal.

Since calculating a MAC is an expensive operation for a microcontroller, Sancus includes an optimisation scheme for subsequent authentications. When successfully enabling isolation for a new SM as discussed above, the processor associates it with a non-zero sequential smID that is exclusively managed by hardware. The hardware implementation ensures that smIDs are not re-used before resetting the platform. That is, smIDs uniquely identify an SM within one boot cycle. A programmer can retrieve smIDs using the `sancus_get_id` instruction. This instruction takes an address as an argument and returns the smID of the module loaded at that address, or zero if no module is enabled at the provided address. The `sancus_verify_address` instruction returns the smID of the verified module on successful MAC comparison. The caller may store this smID to simplify subsequent authentications. It indeed

[1] Note that the initial Sancus architecture, as presented in [40], simply performed a platform reset on a memory access violation. The explanation of the interrupt mechanism described here is based on personal communication with Job Noorman.

suffices to compare the stored smID with the result of the `sancus_get_id` instruction to ensure that the callee module is still loaded at the expected address.

After successfully verifying SM_2 as above, SM_1 can pass any arguments for the call through CPU registers to guarantee confidentiality and integrity of the data, as explained in Sect. 2.2.2.

3.1.5 Caller Authentication

The above secure linking features provide a caller with guarantees about the state and identity of the module that it is about to call. Sancus can also provide a callee with similar guarantees about its caller.[2] To so, Sancus offers the `sancus_verify_caller` instruction that calculates a MAC of the identity of the calling module and compares it with the expected MAC that is provided as an argument. The hardware logic thereby defines the calling module as the previously executing module and provides the `sancus_get_caller_id` instruction to retrieve its smID. Like the secure linking instructions from the previous section, the caller authentication instructions are inherently unforgeable as they are entirely implemented in hardware. Two SMs can therefore efficiently establish a mutually authenticated communication channel without trusting any software.

None of the PMAs discussed in Sect. 2.2.3 provides authentication features as strong or efficient as those of Sancus. Strackx et al. [49] propose an approach to set up a mutually authenticated communication channel without using explicit caller authentication hardware instructions. As explained in Sect. 2.2.2 they use the provided return continuation point to identify and verify the module where execution will continue. Their approach thus guarantees information is returned to the correct module, but cannot reliably identify the originator of the message. An attacker may indeed provide an arbitrary return address if he is interested in effect only. The initial Sancus architecture [40] as well as Trustlite [30] therefore employ a three way handshake protocol to set up a true mutually authenticated communication channel without hardware support for caller authentication. First, the calling module SM_1 verifies the module SM_2 it is about to call, using the techniques from the previous section. Next, SM_1 jumps to the physical entry point of SM_2, passing (*i*) a secret nonce, and (*ii*) the address of its own physical entry point as parameters. SM_2 then uses the techniques from the previous section to verify the module corresponding to the provided address and on success calls back to SM_1, passing the secret nonce as an argument. SM_1 finally responds to SM_2 that it has indeed called SM_2 with that nonce. SM_2 may now safely accept the call. Needless to say that, compared to the simple `sancus_verify_caller` and `sancus_get_caller_id` instructions presented above, such a three way handshake procedure imposes a serious overhead.

As explained further on, the work presented in this master's thesis relies heavily on Sancus' efficient and reliable caller authentication mechanism.

[2] Sancus' caller authentication features are recent work and therefore not discussed in [40]. The explanation provided here is based on personal communication with Job Noorman.

3.1.6 Remote Attestation

Sancus allows a remote software provider to (*i*) verify that its module has been deployed correctly, and (*ii*) communicate securely with that module, preserving confidentiality, integrity and authentication over an untrusted communication channel.[3] The basis for these security guarantees lies in the fact that an SM that was correctly loaded shares a secret cryptographic key with its software provider, as explained in Sect. 3.1.2. Sancus thus offers two more instructions that operate on the key of the currently executing module. The `sancus_wrap` instruction calculates the MAC of a given message and encrypts its content, whereas the `sancus_unwrap` instruction verifies the given MAC of a given message and decrypts its content on success. Since Sancus enforces that a module can only indirectly access its own key via dedicated hardware instructions, a software provider is guaranteed that the message was produced by a specific module that is running uncompromised.

Table 3.1 summarises all the introduced hardware instructions and lists their arguments and return values. The instructions that are marked in the "Crypto" column are slower since they use cryptographic primitives, as explained above.

Table 3.1: Overview of Sancus' extended instruction set

Instruction	Crypto	Arguments	Return Value
`sancus_enable`	✓	SM layout and SP identifier	Success bool
`sancus_disable`	-	-	-
`sancus_verify_address`	✓	Address and expected MAC	smID
`sancus_verify_caller`	✓	Expected MAC	smID
`sancus_get_id`	-	Address	smID
`sancus_get_caller_id`	-	-	smID
`sancus_wrap`	✓	Memory pointers	Success bool
`sancus_unwrap`	✓	Memory pointers	Success bool

3.1.7 Sancus Module Entry and Exit Protocol

A programmer can use the extended instruction set from Table 3.1 to securely write his own SMs in assembly code. That is, all the above security guarantees (memory isolation, secure linking, caller authentication and secure communication) can be accomplished with a zero-software TCB. Sancus also comes with a dedicated C compiler to allow the development of SMs in a higher level language on top of the basic PMA hardware. The compiler reduces SM creation to simple annotation of the C code with `SM_ENTRY`, `SM_FUNC` and `SM_DATA` attributes. This section explains how the compiler takes care of low-level things to ensure that SMs represent an isolated execution environment.

[3] Note that the initial Sancus architecture presented in [40] only ensured integrity and authentication through a hardware-computed MAC of the passed data. The instructions presented in this section also preserve confidentiality and represent recent work on Sancus. The explanation is therefore based on personal communication with Job Noorman.

As explained in Sect. 2.2.2, SMs should maintain their own private call stack to ensure their internal execution cannot be influenced from the outside. When switching protection domains, the compiler should make sure that (*i*) the private call stacks are properly saved/restored, and (*ii*) all unused CPU registers are cleared to avoid private data leakage. To this end, the compiler generates short assembly code stubs that are executed whenever a module is entered or exited.[4] These stubs, referred to as respectively `sm_entry` and `sm_exit`, are inserted into the code section of every SM. The following paragraphs elaborate on their responsibilities.

Entering a Module Recall from Sect. 3.1.3 that SMs have a single physical entry point at the start of their code section. By inserting the `sm_entry` stub at the physical entry point, the compiler can make sure that the stub is executed whenever the SM is entered. This allows the `sm_entry` stub to restore the private call stack. The compiler reserves protected memory for the call stack and allocates a fixed location in the private data section to store the stack pointer. Since the private data section is zeroed out on `sancus_enable`, the base address of the stack is stored at a fixed location in the text section. This allows the `sm_entry` stub to initialise the stack pointer after the module was successfully enabled.

The `sm_entry` stub can also be used to forward multiple logical entry points through the single physical entry point. To do so, the Sancus compiler assigns a logical eIDX identifier to each function annotated with `SM_ENTRY` and generates a private jump table for each SM. When calling a module, the eIDX of the desired logical entry point should be provided in an agreed register. The `sm_entry` stub then simply indexes in its jump table with the provided eIDX to retrieve the internal function address. After calling an external function, a module can be re-entered by supplying a special eIDX value, analogous to the approach described in Sect. 2.2.2.

To successfully call an SM, one should jump to its physical entry point, providing all arguments in agreed CPU registers. The `sm_entry` stub expects three kinds of arguments. The first two – the eIDX identifier and the address where execution should be continued after calling this module – are supplied through fixed caller-saved registers. Moreover, the `sm_entry` stub will forward any arguments intended for the logical entry function, according to normal function calling conventions. The MSP430 architecture uses four 16 bit registers R15 to R12 for function arguments [52], which limits the maximum amount of information to be securely passed for one function call to 64 bits. One may encrypt additional data, store the cypher text in unprotected memory and pass the encryption key securely through the CPU registers.

Exiting a Module The compiler also generates an `sm_exit` stub that will be executed whenever a module calls an external function. This stub will first store the internal execution context (program counter, values of CPU registers) on the

[4] Note that the Sancus paper [40] does not explain these stubs in great detail. They are relevant however for the work on secure scheduling, presented in Chapter 5. The explanation provided here is therefore partly based on the source code, which can be found at [39] (files `sm_entry.s` and `sm_exit.s` in `sancus-compiler/src/stubs/`).

private call stack. Next, it clears all unused CPU registers to avoid leaking of internal private data (i.e. only registers that hold an argument value for the callee are not cleared). The `sm_exit` stub thereafter stores the private call stack pointer in a fixed location in the private data section, so that the `sm_entry` stub can restore the stack on re-entry. Finally, the `sm_exit` stub calls the external module, according the calling conventions explained in the previous paragraph.

When calling an external function the `sm_exit` stub passes the address of the physical entry point of its own module as the return address argument. On re-entry, as detected by the special eIDX logical entry index, the `sm_entry` stub simply restores the private call stack and continues internal execution.

3.2 Trustlite

This section briefly compares the Sancus [40] architecture presented above with Trustlite [30], a similar hardware-level PMA for embedded devices.

3.2.1 Overview

Like Sancus, Trustlite is explicitly targeted at small embedded devices without hardware support for virtual memory or processor privilege levels. Trustlite's attacker model is similar to that of Sancus too: attackers may control all unprotected code and may deploy additional modules of their choice, but do not have physical access to the node.

SMs are called *trustlets* in Trustlite and represent isolated execution environments in the shared address space. Trustlets are provided with the typical PMA security guarantees from Sect. 2.2.2: they can only be entered from a few predefined entry points and can be provided with exclusive access to their private data section. Moreover, trustlets running on the same node can inspect each others state and set up a mutually authenticated communication channel.

The Trustlite architecture also features a modified hardware exception engine. In short, this secure exception engine stores the execution context of a trustlet on the corresponding private call stack before jumping to the untrusted ISR. This ensures that even in the presence of interrupts, a trustlet solely depends on hardware for integrity and confidentiality of its internal state. Such a set-up allows an untrusted preemptive scheduler to schedule trustlets in between normal unprotected tasks. Section 5.1.2 discusses Trustlite's task model in more detail.

3.2.2 Execution-Aware Memory Protection Unit

The main difference between Trustlite and Sancus is how they enforce the program counter based memory access control rules. Recall from Sect. 3.1.3 that Sancus employs a dedicated MAL hardware circuit per SM to enforce memory isolation. Trustlite on the other hand features an Execution-Aware Memory Protection Unit (EA-MPU). Such an EA-MPU allows to program access control rules in a dedicated fixed-sized hardware table. An entry in this table looks as follows:

(subject address range, object address range, r/w/x permissions)

The hardware logic verifies on every memory access whether the address being accessed is part of an object range. If so, it uses the value of the current program counter to identify the subject performing the memory access and its corresponding access rights on the object. Recall that Sancus' hardware logic provides the code section of an SM with exclusive access to a single contiguous data section. Trustlite's EA-MPU table on the other hand can encode more complex policies. Think for example about multiple non-contiguous private data sections per trustlet or protected shared memory between trustlets. Such flexibility is however hard-limited by the number of entries in the EA-MPU table, which is defined at hardware synthesisation time.

Recall from Sect. 3.1.2 that Sancus supports a dynamic deployment model where a remote software provider can load or unload SMs at run time without trusting any software on the node. Trustlite on the contrary presupposes a static deployment model where trusted software, referred to as the Secure Loader, loads and protects all desired trustlets on system boot. More specifically, the Secure Loader is responsible to (*i*) load all desired trustlets in memory, (*ii*) fill in the desired memory access rules for trustlets in the EA-MPU table, and (*iii*) fill in a special rule that renders the EA-MPU table read-only. From this moment on, the encoded access rules are enforced by the EA-MPU hardware and the Secure Loader may safely transfer control to the untrusted OS. The Secure Loader is obviously part of the TCB for trustlets, since they have no way of reliably verifying whether their protection rules or code section has changed during the loading process. Trustlets therefore rely on the hardware for access control enforcement and on the Secure Loader to set up things correctly. This explains why Trustlite, in contrast to Sancus, does not feature an extended hardware instruction set, such as the one from Table 3.1.

A final aspect of Trustlite's design concerns secure linking. Recall from Sect. 3.1.4 that Sancus uses hardware-managed keys and dedicated instructions to ensure strong SM authentication guarantees. Trustlite on the contrary provides secure linking through the read-only nature of the EA-MPU table as follows. First, the calling trustlet looks up the callee in the EA-MPU table to verify it is indeed loaded at the correct address with protection rules enabled. Next, the calling trustlet may calculate a cryptographic hash of the code section of the callee to verify its integrity. Note that verifying a hash of the callee does not exclude the Secure Loader from the TCB since he may simply change the hash value stored in the calling trustlet's code section. Trustlets can set up a mutually authenticated communication channel through a three way handshake protocol, as explained in Sect. 3.1.5.

3.2.3 Trusted Computing Base

Trustlite and Sancus share the same goal, i.e. providing lightweight hardware-enforced protection domains in a single-address-space, but their approach differs significantly in terms of flexibility and the induced TCB. Sancus realises strong isolation and authentication guarantees with a zero-software TCB in a dynamic

deployment environment, but offers limited flexibility. That is, the access control policies for SMs are hard-wired in the combinational MAL circuits. Trustlite on the other hand allows to encode more flexible access control policies in its EA-MPU table, but relies on an omnipotent Secure Loader software entity for initialisation and does not support a dynamic deployment model.

Trustlite's EA-MPU rules are enforced by hardware, but initialised by a trusted software layer. Trustlite's Secure Loader therefore resembles a traditional OS (micro)kernel that is inevitably part of the TCB. In this respect, the initial Trustlite architecture and static deployment model only depend on the Secure Loader at initialisation time. Koeberl et al. [30] however also mention the possibility of introducing a software entity that (un)loads trustlets at run time, reconfiguring the EA-MPU as needed. They acknowledge however that such an entity "must have a notion of existing tasks and mediate IPC" and that "in this case the trust relationships are similar to those of a microkernel OS" [30].

In contrast to Trustlite, the Sancus architecture enforces its security guarantees without trusting any software running on the node. Its hardware-enforced protection scheme indeed makes an omnipotent kernel layer, however small, inherently impossible. This raises the question of what an OS for a Sancus-like hardware-only PMA should look like. The next section therefore discusses the consequences of a zero-software TCB on OS design.

3.3 Operating System Support for Protected Modules

The above discussion has revealed that PMAs are closely related to OS design. The OS is traditionally regarded as a trusted resource manager that separates processes, guards their interactions and implements some form of access control for shared system resources (e.g. CPU time, memory, peripheral devices, etc.). PMAs that isolate fine grained protection domains in a single-address-space therefore adopt some of the responsibilities of a traditional OS. In this respect, the PMAs [48, 3, 36] from Sect. 2.2.3 all employ program counter based access control as a way to complement the existing coarse-grained virtual memory protection scheme enforced by the OS. A lightweight hardware implementation of protected modules on the other hand can provide a standalone substantive memory protection scheme for low-end embedded devices. The Sancus [40] and Trustlite [30] architectures, introduced above, indeed reorganise the unprotected single-address-space into a set of hardware-enforced protected modules. Sancus thereby explicitly excludes the OS from the TCB and Trustlite only relies on a Secure Loader software entity to set up things correctly. Protected modules from both architectures thus exclusively rely on hardware to enforce memory access control rules at run time.

Excluding the OS kernel from the TCB allows for strong security guarantees, but also secludes modules in their own protection domain. The next sections therefore explore the idea of securely providing OS-like services to protected modules, without introducing an omnipotent kernel.

3.3.1 Self-Protecting Operating System Modules

As already stated above, a hardware-level PMA secludes SMs in their own protection domain. That is, an SM should either fulfil its own needs or rely on the services of an untrusted OS to interact with the outside world. This implies poor trade-offs between flexibility and protection. Consider for example an SM that wants to save confidential data in a file system or read secret values from a sensor. Without additional support, this SM would have to either claim the file system/sensor for itself, effectively denying others access to the resource, or accept to use it in an unprotected way. Both options are obviously undesirable in an embedded context where system resources are scarce and SMs want to retain the confidentiality and integrity of their data.

The key idea explored in this master's thesis is to overcome the above flexibility vs. protection trade-off by complementing the hardware-enforced security guarantees for SMs with software-based resource access control guarantees. More specifically, this thesis builds upon the existing Sancus primitives to provide a dedicated module SM_{server} with exclusive access to a system resource and implements a thin software layer on top to enforce flexible SM-grained access control policies. Sancus' hardware logic ensures that SM_{server} is solely responsible for the resource it encapsulates. Sancus' unique caller authentication mechanism, discussed in Sect. 3.1.5, furthermore allows SM_{server} to reliably implement access control as desired. The SM_{server} module is however in no way more privileged and cannot undo the hardware-enforced security guarantees of its clients. This shows that even though SM_{server} performs a typical OS task – i.e. shared resource management – it differs significantly from a conventional omnipotent trusted kernel software layer.

Secure resource sharing for PMAs thus requires a disjoint set of *self-protecting OS modules*. Every such module encapsulates and controls access to a platform resource (e.g. a protected memory buffer, a file system, a keyboard, a network interface, etc.). Client SMs can use Sancus' strong authentication mechanism, discussed in Sect. 3.1.4, to establish explicit trust relationships with the self-protecting OS modules of their choice. This allows client SMs to extend their hardware-enforced security guarantees with software-based availability and access control guarantees for shared system resources.

3.3.2 Zero-Software Microkernel

The above idea of implementing the OS as a set of non-privileged modules echoes the widely known microkernel approach [33, 34, 50]. Recall from Sect. 2.1.2 that a microkernel architecture minimises the size of the trusted kernel software layer by implementing most of the OS's functionality as ordinary user space server processes. Since processes are confined in their own virtual address space, a misbehaving OS server cannot harm other applications running on the system. A microkernel design therefore limits the TCB by reducing the kernel's size.

There is no consensus on which mechanisms should be implemented in the microkernel. In this respect, Liedtke [33] reasons about the minimal set of abstractions

that a software microkernel should provide to allow the correct functioning of a conventional virtual-memory-based computer system. He thereby defines correctness as the ability to create arbitrary independent subsystems that can securely contact each other. While Liedtke presupposes hardware support for virtual memory, these abstract "subsystems" clearly resemble SMs. The Sancus platform could therefore be regarded as a truly minimal zero-software microkernel that provides two basic mechanisms to SMs: memory isolation and authentication. The question then becomes whether such a zero-software microkernel is sufficient to securely implement OS-like services on top. Liedtke [33, 34] identifies only three software primitives for his minimalist second generation L4 microkernel: address spaces, threads and IPC. The following briefly compares these microkernel abstractions to Sancus' hardware-enforced mechanisms:

Address Spaces Liedtke argues that a microkernel has to "hide the hardware concept of address spaces, since otherwise, implementing protection would be impossible." [33]. That is, a microkernel should maintain the mapping of virtual to physical addresses for each process. Sancus on the other hand provides fine-grained hardware-enforced protection domains in a single-address-space, as explained in Sect. 3.1.3.

Threads Liedtke identifies a thread as "an activity executing inside an address space" and states that "to prevent corruption of address spaces, all changes to a thread's address space [...] must be controlled by the kernel" [33]. He furthermore mentions support for preemption as an additional reason to include a threading concept in the kernel. The Sancus architecture on the other hand does not need the concept of a classical thread to realise its security guarantees. As explained in Sect. 3.1.5, Sancus' hardware logic does however include a notion of control flow by identifying the currently and previously executing SM. The work presented in Chapter 5 shows that this is a sufficiently strong mechanism to realise secure multithreading in a single-address-space, but also identifies the need for a secure hardware interrupt engine in Sancus.

Inter Process Communication Liedtke identifies the need for a microkernel to "establish a communication channel which can neither be corrupted nor eavesdropped" and states that "[unique identifiers] are required for reliable and efficient local communication" [33]. This clearly resembles Sancus' hardware-supplied unique smIDs and secure linking features, discussed in Sect. 3.1.4.

The above comparison illustrates how Sancus' hardware primitives resemble the abstractions implemented by a minimal trusted microkernel. This seems to indicate that Sancus provides sufficiently strong hardware building blocks to securely allow the software implementation of OS-like services on top. Like in the microkernel analogy and as mentioned above, this requires regarding the OS as a set of unprivileged modules that realise policies not offered by the hardware. The key thing to note here however is that Sancus realises its security guarantees without a trusted kernel software layer.

This master's thesis focuses on providing secure resource sharing for a Sancus-like hardware-level PMA. Chapter 4 presents a generic access control scheme and implements a protected file system SM_{sfs} module as a case study of realising SM-grained logical file access restrictions. SM_{sfs} offers security guarantees similar to those of user space file system server which is effectively shielded from other protection domains. Chapter 5 thereafter presents a multithreading model for hardware-level PMAs and implements an accompanying unprivileged secure scheduler SM_{sched} module. The work presented in this master's thesis therefore shows that SMs can be provided with typical OS guarantees, without loosing their hardware-enforced security guarantees.

3.4 Conclusion

A lightweight hardware-level PMA is promising in an embedded context, but also secludes SMs in their respective protection domains. That is, hardware-enforced PMAs offer strong memory isolation and authentication guarantees, but cannot natively realise flexible access control policies for shared system resources. Without additional support, the only way for an SM to get availability and/or confidentiality and integrity guarantees for a system resource is to claim the resource for itself. This implies poor flexibility vs. protection trade-offs, especially in an embedded context where system resources are scarce and should be shared among multiple inter-untrustworthy software entities. Think for example about an SM that denies others access to a MMIO flash drive to protect its own confidential data or an SM that monopolises CPU time to be guaranteed availability at all time.

It would thus be valuable to supplement the hardware-enforced security guarantees for SMs with OS-like software-based access control policies for shared system resources. As in the microkernel analogy, this requires implementing core OS concepts in unprivileged modules that make use of the Sancus-provided primitives. The challenge is to this without introducing an omnipotent kernel software layer that invalidates the hardware-enforced security guarantees for SMs. The following two chapters provide two case studies in this respect. Chapter 4 discusses a generic access control model and implements a thin protected file system access control software layer that provides logical file protection guarantees for client SMs. Chapter 5 presents a threading model for hardware-level PMAs and implements an accompanying secure scheduler that provides logical threads with CPU availability guarantees.

Chapter 4

Logical File Access Control

This chapter presents a protected file system for the Sancus platform [40]. The file system is encapsulated in its own SM_{sfs} protection domain with exclusive access to the storage device, ensuring file system integrity and confidentiality. Furthermore, it realises *SM-grained access control*, allowing fine-grained access control policies for logical file sharing between SMs. In a wider context, the prototype therefore demonstrates the feasibility of encapsulating and controlling access to a shared system resource through a lightweight trusted software layer on top of hardware-enforced mechanisms.

The discussion is organised as follows. First, Sect. 4.1 motivates the need for a secure embedded file system. Section 4.2 thereafter presents the protected file system SM_{sfs} implementation and Sect. 4.3 evaluates its runtime overhead. Section 4.4 discusses the security guarantees and limitations of the prototype and compares the approach with other PMAs. Section 4.5 finally concludes.

4.1 Motivation

Supplementing the hardware-enforced security guarantees of SMs with logical file access control guarantees is valuable for two major reasons. First, existing embedded file systems commonly lack support for efficient file protection, as discussed in Sect. 4.1.1. Second, providing SMs with the concept of a protected logical file has many application areas, as introduced in Sect. 4.1.2.

4.1.1 Embedded File System Security

Existing embedded file systems [20, 19] focus mainly on performance aspects: flash specific optimisations, RAM usage and energy consumption, whereas file protection is non-existing or remains very limited. This is in line with the original concept of a single static unprotected embedded application. Indeed, the design notes for Matchbox, a file system for TinyOS, state literally: “We do not need: Security in any form, [...]” [22]. As another example, Contiki features the Coffee file system [51], a dedicated lightweight flash file system without any form of access control. LiteOS [6]

provides its own LiteFS UNIX-like file system in which files may represent data, binaries or devices. It also offers a coarse-grained user-oriented protection mechanism that classifies all users in one of three levels, each with its own `rwx` mode bits.

In an embedded context, featuring a dynamic multi-stakeholder deployment model, it is software modules rather than users that represent the unit of file protection. Indeed, recall from Sect. 3.1.3 that an SM represents the unit of memory protection and authentication. Extending these guarantees with SM-grained file access control would thus be valuable. File protection on a per-SM-basis would furthermore be interesting as it differs from conventional UNIX-like user-oriented file protection [4]. UNIX decides file access based on the identity of the owner of the currently executing program. This coarse-grained scheme does however not shield a user from malicious programs that run with her permissions [5]. Moreover, fine-grained file protection is hindered by the default `owner/group/others` file attributes. Capability-based process-specific file protection for UNIX has been proposed [5] as a countermeasure and fine-grained access control can be accomplished with access control lists [23].

4.1.2 Application Scenarios

The problem domain of low-end embedded devices is characterised by conflicting interests between economic considerations on the one hand and security requirements on the other. PMAs present the SM as the unit of lightweight memory isolation and authentication. The protected SM_{sfs} file system module introduced in this chapter shows the feasibility of securely sharing system resources on a per-SM-basis. More specifically the file system can control access to (*i*) a private memory buffer, allowing a form of protected shared memory between SMs, and (*ii*) a peripheral flash drive, allowing protected bulk storage in a real-world embedded file system. The following briefly discusses application scenarios for both possibilities.

Protected Shared Memory

Being able to pass a moderate sized buffer securely between protection domains, could be useful in many contexts. A first scenario concerns parameter passing of large values. Indeed, recall from 3.1.7 that one can only provide parameters securely through a limited number of CPU registers when calling an SM. The only way to pass large values securely between SMs is therefore to either (*i*) encrypt the data in unprotected memory and pass a pointer to it, or (*ii*) call the destination module multiple times, each time passing part of the data securely through registers. The SM_{sfs} module can facilitate this second process by acting as a trusted third party that temporarily stores all transited data in its own private section and allows fine-grained access rights for the recipients.

Protected shared memory may also be useful in the context of secure I/O. Recall from Sect. 3.1.3 that an SM can indeed be provided with exclusive access to a MMIO peripheral. Think about such a keyboard driver module $SM_{keyboard}$, offering an entry function to get an input line confidentially from the user. $SM_{keyboard}$ may simply use SM_{sfs}'s protected shared memory service to store the potentially large

result and to define according access rights for the client that requested the data. In this context, it is useful to think about more complicated scenarios where multiple SMs work on a common shared memory buffer. Consider for example a client SM_A requesting an input line from a user input framework SM_{input}. This module in turn uses $SM_{keyboard}$ to get the actual input line through shared memory, potentially makes some changes (e.g. auto completion, expansion, etc.) and finally returns the result to SM_A. Furthermore, these modules could have different access rights to the shared buffer: e.g. `write` for $SM_{keyboard}$, `read/append_only` for SM_{input} and `read/destroy` for SM_A.

Secondary Storage

Several authors [20, 19, 51] identify an emerging application area for embedded platforms using secondary storage file systems. As explained in Sect. 4.1.1 support for logical file protection in existing embedded file systems remains very limited. In a multi-stakeholder model with software extensibility by multiple untrustworthy vendors however, fine-grained access control for secondary storage resources is essential. Consider for example a low-end extensible wearable device. One application could save sensitive medical logs in the file system; another one could simultaneously use the file system to save privacy-sensitive data such as environment sensor data, recordings, GPS locations, etc. Needless to say reliable and fine-grained memory protection and file access control is imperative in such a system.

4.2 Design and Implementation of a Protected File System

This section presents a generic access control mechanism and accompanying protected file system implementation for the Sancus platform [40]. The file system is encapsulated in its own SM_{sfs} protection domain with exclusive access to the back-end device, which can be either a protected memory buffer or a real-world embedded flash file system. It features a thin access control software layer that allows secure logical file sharing between SMs. The protected file system therefore serves as a case study of secure resource sharing in a PMA environment.

The explanation is organised as follows. Section 4.2.1 shortly reviews an initial inode-based file system prototype. Thereafter, Sect. 4.2.2 explains how the current SM_{sfs} module is internally structured into a generic front-end that controls access to a pluggable private back-end software layer. Section 4.2.3 introduces the front-end's implementation and Sects. 4.2.4 and 4.2.5 discuss the respective back-ends.

4.2.1 Initial Inode-based File System Prototype

Initially, an elementary inode-based file system prototype was implemented from scratch to explore the feasibility of encapsulating it entirely in its own protection domain. Analogous to a conventional UNIX structure [4, 50], this file system stores logical file meta data – size and permissions – in the on-disk inode table. To do

so, the implementation features a custom inode layout that includes a fixed-sized Access Control List (ACL) to store the permissions. By storing access control meta data in the on-disk inode table, this prototype anticipates SM-grained file protection that persists across system reboots. As further discussed in Sect. 4.4.3, such persistent protection guarantees are however non-trivial and require i.a. persistent SM authentication, which is not available on the current Sancus hardware.

This primitive file system demonstrates the feasibility of SM-grained logical file protection, but its internal structure – lacking advanced caching or flash-specific optimisations – is too limiting in the context of real-world embedded applications. The next sections therefore discuss an approach that separates the non-persistent access control logic from the actual file system implementation to allow logical file protection guarantees for a real-world embedded flash file system.

4.2.2 Layered Design

The protected file system depicted in Fig. 4.1 features a layered design with a *front-end* access control layer deciding access to a private *back-end* software layer, encapsulating the actual resource. From the point of view of the front-end, the back-end is an abstract Contiki File System (CFS) interface [16] implementation that can be plugged in when compiling the SM_{sfs} module. This chapter provides two different back-end implementations. Section 4.2.4 discusses an implementation that operates on a Sancus-protected memory buffer, allowing a form of protected shared memory between SMs. Section 4.2.5 plugs in a real-world embedded flash file system, realising SM-grained protection for a shared system resource.

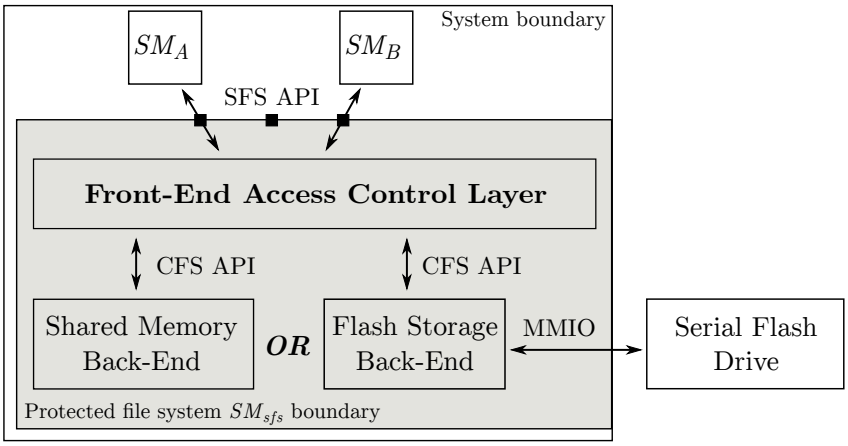

Figure 4.1: The protected file system SM_{sfs} module consists of a generic public front-end access control layer controlling access to a pluggable private back-end software layer, encapsulating the actual resource.

From a security perspective, the front- and back-ends are merely a logical structure since the entire file system runs in a single protection domain SM_{sfs}. The front-end offers the public interface (i.e. SM_{sfs}'s entry points) towards clients, whereas the

back-end is called through private non-entry functions. Since the PMA hardware guarantees a protection domain can only be entered from its predefined entry points, a client is effectively prohibited from bypassing the access-control front-end and calling the back-end directly.

The division of responsibilities between the front- and back-end is as follows. The front-end presents a transparent UNIX like file system interface towards client SMs to provide them with the concept of a contiguous logical file with offset-addressable content. Internally however, the front-end is only concerned with SM-oriented access control policies and maintains the data structures to do so. It relies on the back-end CFS implementation for the concept of a logical file. The back-end in its turn encapsulates the actual file system implementation and is completely unaware of any access control going on. It is important to note here that the front-end has no notion of persistence and stores all its access control data structures in volatile protected memory. As further discussed in Sect. 4.4.3, the SM_{sfs} prototype does not support persistent SM-grained file protection since it uses Sancus' hardware-provided smIDs that do not last over multiple boot cycles [40].

4.2.3 Generic Front-End Access Control Layer

The front-end is conceived as a wrapper implementation that associates an ACL of `(smID, permissions_flag)` pairs per logical file to validate the caller's permissions before passing the call to the back-end.

Software-Module-Grained Access Control

Recall from Sect. 3.1.5 that the smIDs, uniquely identifying a Sancus module within one boot cycle, are inherently unforgeable as they are exclusively managed by hardware. They can therefore safely be used for subsequent client authentications in a software layer. Essentially, the front-end accomplishes its access control guarantees through the `sancus_get_caller_id` hardware instruction, which it uses to reliably retrieve the smID of the client – i.e. the SM that entered the currently executing module.

The protected file system prototype SM_{sfs} builds upon Sancus' hardware-enforced security guarantees in two ways. On the one hand, Sancus' memory isolation techniques grant SM_{sfs} exclusive access to its back-end resource. On the other hand, Sancus' SM identification scheme provides SM_{sfs} with a reliable client authentication mechanism that allows implementing a thin software layer to realise flexible access control policies for its private back-end resource.

Interface

The Sancus File System (SFS) interface is based upon the UNIX-like Contiki File System (CFS) interface [16, 51], modified where needed and extended with SM-specific access control functions. More specifically, the `cfs_read` and `cfs_write` functions that require a pointer to an unprotected memory buffer and a length argument, are replaced with `sfs_getc` and `sfs_putc` functions, which pass the

arguments and return values securely through CPU registers. For the same reason file name strings are replaced with single chars.

In addition, the SFS interface supports SM-grained access control. Using the `sfs_chmod` function, the module that first created a file can assign or revoke fine-grained permissions for a specific SM via a bit flag. Currently the prototype supports read-only, write-only and read-write permissions, but due to the generic access control scheme, more advanced policies such as append-only could be added relatively easy. Client SMs open files through a modified `sfs_open` function, requiring a permissions flag argument and an initial size hint which is passed to the back-end.

Data Structures

The SM_{sfs} prototype stores all access control data structures in its protected private data section. It maintains a linked list for logical files, each with a corresponding SM-grained permission ACL. This allows a two phase permission lookup procedure when specifying a file by name. The file list is first traversed to locate the file, using the name as a key. Thereafter, the corresponding ACL is searched using the calling SM's smID as a key. To speed up future accesses using a file descriptor, SM_{sfs} employs a fixed-sized file-descriptor-indexed array with pointers to the corresponding ACL entry.

On each function call, before translating the call to the CFS back-end, the front-end validates the caller's permissions. If the caller passes a file descriptor, the implementation first checks whether it is in the expected range and points to an ACL entry that belongs to the caller. Furthermore, to allow safe revocation of earlier assigned permissions, SM_{sfs} closes any remaining open file descriptors when revoking a permission – as opposed to the POSIX standard [26] which leaves such behaviour implementation-defined.

As explained in Sect. 3.1.3, Sancus' hardware logic requires the protected memory section of a module to be fixed-sized during its lifetime. The SM_{sfs} implementation should fulfil its own dynamic protected memory requirements. To do so, the implementation enforces a maximum number of open file descriptors, pre-allocates a fixed number of file and permission structs at compile time and maintains them in a free list at run time. When running out of protected memory, the front-end rejects requests to create additional files.[1]

4.2.4 Protected Shared Memory Back-End

In the protected shared memory implementation, the back-end operates on a fixed-sized Sancus-protected memory buffer. Internally, SM_{sfs} uses a dynamic memory allocation `malloc` implementation on this buffer, allowing clients to transparently claim a portion of it through a familiar UNIX-like file system API.

[1] Note this allows denial-of-service attacks, where an attacker deploys an SM that creates the maximum number of files. This could be mitigated in turn by enforcing more complex policies, such as limiting the maximum number of open files per SM. Availability is discussed in more detail in Sect. 4.4.2.

Logical files in the protected shared memory back-end have a fixed size during their lifetime. When creating a new file, the implementation uses the initial size argument to allocate a buffer of the corresponding size. From then on, it does proper bounds checking, refusing to seek beyond the buffer's end.

Files are arranged in a linked list, each i.a. containing a pointer to a location inside the private `malloc` buffer and the corresponding size. Furthermore, as in the front-end, the prototype maintains a file-descriptor-indexed array to speed up common file operations and to save the current client-specific logical offset in the file. Of course this bookkeeping information should also reside in protected memory. To support a dynamic number of logical files, the implementation allocates the required structs using its own protected `malloc` buffer.

4.2.5 Protected Shared Flash Storage Back-End

The case study flash file system back-end is Contiki's open source Coffee FS [51]. The reasons for this choice are that the Coffee file system is (*i*) highly optimised for small flash memories, (*ii*) requires a small and constant RAM footprint per open file, and (*iii*) does not provide any existing file protection mechanism.

The shared flash storage back-end introduces the important issue of *secure peripherals* [30]. Indeed, SM_{sfs} should be provided with exclusive access to the flash drive to ensure file system integrity and confidentiality. For peripherals that are being accessed through the memory address space, Sancus' program counter based memory access control scheme grants a dedicated driver SM exclusive access to a resource by including the relevant MMIO addresses in its private data section [40]. The driver module then securely links with SM_{sfs}, using mutual authentication as discussed in Sects. 3.1.4 and 3.1.5, to realise end-to-end file system protection.

4.3 Experimental Evaluation

This section evaluates the protected file system SM_{sfs} prototype introduced above. It discusses runtime overhead as well as the induced memory footprint and code size. Total runtime overhead is defined from a client SM's perspective as the additional number of CPU cycles needed to call an SM_{sfs} entry function, compared to calling the respective function of an unprotected file system. Sections 4.3.1 and 4.3.2 split the overall overhead into a Sancus-dictated component, induced by switching Sancus protection domains, and an implementation-dependent component caused by the access control layer. Finally, Sects. 4.3.3 and 4.3.4 provide the relative overhead for the protected shared memory and Coffee flash file system back-ends.

All experiments were conducted on a Sancus-enabled MSP430 FPGA running at 20 MHz. The FPGA is connected to a Micron M25P16 serial flash drive, using the Coffee file system from Contiki release 2.7.

4.3.1 Sancus Protection Domain Switching

As explained in Sect. 3.1.7, the Sancus compiler automatically inserts `sm_entry` and `sm_exit` code stubs that take care of private call-stack switching and clearing of CPU registers to avoid leaking of confidential data. These code stubs thus induce an extra overhead for function calls that switch protection domains. The exact number of cycles needed for such a function call varies with the number and size of the arguments and return value. Calling an unprotected function from within a module SM_A takes between 120 and 170 cycles, whereas calling an SM_B entry function from within SM_A requires between 210 and 280 cycles. The latter interaction consumes more cycles since SM_B's `sm_entry` stub additionally has to (*i*) store all callee-save general purpose CPU registers, (*ii*) call the internal logical entry point, (*iii*) restore all callee-save general purpose CPU registers, and (*iv*) clear registers that do not hold a return value.

The above results indicate an additional Sancus-dictated overhead of roughly 100 cycles for client SMs calling our protected SM_{sfs} module, as opposed to calling an unprotected file system. Note that this overhead is solely caused by encapsulating the file system in its own protection domain SM_{sfs}, independent from any additional access control logic.

4.3.2 Access Control Overhead

This section provides micro benchmarks of the access control front-end layer. The last column of Table 4.1 shows the total number of CPU cycles needed for a protected client SM_A to call the protected file system SM_{sfs} configured with a dummy back-end that simply returns to the front-end. The "Sancus Induced" column lists the number of cycles thereof caused by calling the respective Sancus entry function, depending on the number of arguments. These numbers are clearly responsible for the vast majority of cycles, illustrating how SM_{sfs} realises SM-grained access control policies through a thin software layer on top of Sancus' hardware-enforced mechanisms.

To further detail the overhead induced by the front-end, the "back-end call" column of Table 4.1 lists the number of cycles needed by the front-end to call the back-end – the downside of a layered design. The "ACL checks" column shows the number of cycles needed to traverse the access control data structures, in the case of a single file and ACL entry. The impact of the file descriptor cache is clearly visible, resulting in a constant and low access control overhead for the functions `seek`, `getc`, `putc` and `close`. As explained in Sect. 4.2.3, the prototype uses linked lists, resulting in a linear growing access control overhead for functions without a file descriptor. As experimentally verified, the worst-case overhead indeed grows linearly with a reasonable factor of about 12 extra cycles per additional logical file or ACL linked list entry.

The memory overhead of the SM_{sfs} prototype is bounded at compile time by pre-allocating the file descriptor array and a maximum number of structs for logical files and ACL entries. This is common practice in embedded file systems – as

TABLE 4.1: The number of cycles needed for SM_{sfs} configured with a dummy back-end, assuming a single open file with one ACL entry. The "Sancus Induced" column lists the number of cycles needed to call the respective SM_{sfs} entry function. The next two columns show the overhead of the front-end and the last column lists the summation.

SFS API		**Sancus**	**Front-End Induced**		**Total**
function	case	**Induced**	ACL checks	back-end call	
`format`		211	181	17	409
`open`	creat	279	120	69	468
`open`	exist	259	95	69	423
`seek`		259	18	58	335
`getc`		229	46	59	334
`putc`		234	55	63	352
`close`		229	56	24	309
`remove`		226	138	27	391
`chmod`	add	247	120	0	367
`chmod`	revoke	247	158	0	405

illustrated by the Coffee back-end – and is needed to overcome Sancus' fixed-sized private data section, as explained in Sect. 4.2.3. Both structs occupy 6 bytes. In the test set-up, the SM_{sfs} module was configured with a maximum number of 10 ACL entries, 5 files and 8 file descriptor entries, resulting in a total memory usage of 106 bytes. In terms of code size, the access control layer of SM_{sfs} occupies 1.9 KB, whereas the Coffee back-end requires 5.3 KB. The front-end access control layer thus increases the code size with a factor of 0.36.

4.3.3 Protected Shared Memory Back-End

To investigate the runtime overhead of the protected file system module SM_{sfs} configured with a shared memory back-end, this section compares it to the case where two SMs communicate via an unprotected dynamically allocated shared memory buffer in the single-address-space. The "shm" column of Table 4.2 thus shows the baseline, i.e. the number of cycles needed to create a shared buffer of size 100 via an unprotected `malloc` call, read / write a character and `free` it. The next two columns list the number of cycles needed for the protected shared memory SM_{sfs} module and the absolute overhead.

The key thing to note here is that, once the unprotected dynamic memory is allocated, read and write accesses are equivalent to normal memory accesses and thus require very few cycles. The SM_{sfs} protected shared memory set-up however adds a level of indirection, implying a huge relative overhead for memory accesses. Moreover, setting up the memory buffer takes longer as the meta data structures should be initialised and clients have to open the logical file before accessing it. Emulating flexible access control policies on top of Sancus' native protection model is however

for the moment the only way of realising complex protected interactions between SMs.

Table 4.2: The overhead for a client SM_A that uses SM_{sfs}'s services for each back-end, assuming a single open file with one ACL entry. The "Shared Memory" columns list from left to right: the number of cycles needed by SM_A to use unprotected dynamic memory, SM_{sfs} with a shared memory back-end and the absolute overhead. The "Flash Storage Back-End" columns list from left to right, the number of cycles needed for SM_A to call: an unprotected Coffee file system, SM_{sfs} with a Coffee back-end; the absolute and relative overhead and the overhead percentage induced by the ACL lookup.

API		**Shared Memory**			**Flash Storage Back-End**				
		baseline		*overhead*	*baseline*		*overhead*		
function	case	shm	sfs-shm	shm-abs	coffee	sfs-coffee	abs	rel	acl
`format`		-	584	584	360 E6	360 E6	286	0	63
`open`	creat	192	1,326	1,134	76,133	76,436	303	0	40
`open`	exist	-	706	706	2,604	2,862	258	10	37
`seek`		-	322	322	430	594	181	44	10
`getc`		2	342	340	902	1,081	179	20	26
`putc`		4	351	347	1,288	1,485	197	15	28
`close`		-	539	539	317	498	181	57	31
`remove`		192	670	478	8,033	8,293	260	3	53
`chmod`	add	-	367	367	-	367	367	-	33
`chmod`	revoke	-	405	405	-	405	405	-	39

4.3.4 Protected Shared Flash Storage Overhead

This section investigates the runtime overhead of the protected SM_{sfs} file system prototype on top of Contiki's Coffee FS [51], a typical real-world embedded flash file system. The "coffee" column of Table 4.2 lists the baseline, i.e. the total number of CPU cycles needed for a protected client SM_A to call an unprotected Coffee flash file system. The "sfs-coffee" column shows the number of cycles needed by SM_A to call the SM_{sfs} protected file system module, configured with a Coffee back-end. Note that these numbers reflect the ideal case where the front-end as well as the back-end implementation and flash driver share the same protection domain SM_{sfs}. In the test set-up the Coffee file system and the flash driver operate in unprotected mode, see also Sect. 4.4.3. The presented data is therefore extrapolated by carefully subtracting the fine-grained overhead of switching Sancus protection domains.

The "abs" column of Table 4.2 lists the absolute number of overhead cycles caused by the protected file system implementation, as compared to the unprotected Coffee set-up. To make sense out of these numbers, the next columns provide the relative overhead and the percentage of the total overhead that is caused by the access control front-end implementation. These results indicate that the overhead of protected

resource sharing on top of a real-world flash file system is reasonable. Indeed, due to the delay of the flash I/O and the file descriptor cache described above, the relative number of additional cycles remains limited for commonly used file operations, under 20 % for `getc` and `putc`, and even drops to zero for file system heavy operations such as `format`, `creat` and `remove`. Moreover, the additional SM-specific `chmod` access control function consumes a number of cycles of the same magnitude as the unprotected in-memory file operations, such as `seek`. Finally, the front-end access control software layer shows to be lightweight in the sense that over half of SM_{sfs}'s overhead – in the case of a single file and ACL entry – can be attributed to calling the respective Sancus entry function and the back-end function call.

Comparing the two case study back-end reveals another characteristic of SM interactions: the relative overhead of switching protection domains from a calling module, decreases with the amount of work done in the callee module. Indeed, the overhead of SM_{sfs} with a flash back-end is lightened by the flash I/O delay, whereas the overhead in the protected shared memory case is aggravated by the fast unprotected memory access.

4.4 Discussion

This section discusses the security/availability guarantees and limitations of the protected file system. The prototype is also regarded in a broader perspective by discussing the possibility of porting the access control approach to other PMAs or controlling access to other peripherals.

4.4.1 Security Guarantees

The SM_{sfs} module builds upon Sancus' existing hardware primitives to supplement the hardware-enforced security guarantees of its clients with logical file access restrictions. Clients using SM_{sfs} naturally incorporate it in their TCB. This approach differs significantly from a traditional trusted OS computing base however for two major reasons.

First, *only* client SMs using SM_{sfs} should trust it and Sancus offers a strong authentication mechanism to verify SM_{sfs}. Recall from Sect. 3.1.5 that a client can indeed attest SM_{sfs} has not been tampered with and was loaded correctly, with exclusive access to the MMIO flash drive addresses. This results in a minimal explicit TCB, as opposed to the implicit TCB induced by an omnipotent kernel trusted software layer.

Second, the SM_{sfs} module is solely entrusted its dedicated file system task, echoing the well-known principle of least privilege [44]. A faulty SM_{sfs} module can indeed only tamper with or leak the file system data it is entrusted. A client SM still preserves exclusive access to its private section. In this, SM_{sfs}'s security guarantees are similar to those of a microkernel file system running in user space as it is effectively shielded from other protection domains. The key thing to note here, is that Sancus does not rely on any trusted kernel software layer to enforce this separation, as discussed in Sect. 3.3.2.

4.4.2 Availability Guarantees

Encapsulating a file system in its own protection domain is not only valuable from a security perspective, but can also offer additional availability guarantees. Clients can encrypt and sign data themselves before storing it in an unprotected file system to be guaranteed confidentiality and integrity, but such an approach still does not guarantee availability. Indeed, nothing stops an attacker from overwriting data in the unprotected file system or from claiming the relevant MMIO regions to deny others access to the flash drive. The SM_{sfs} module on the other hand guards the access to the flash drive and can therefore enforce availability requirements as desired.

In this respect, Masti et al. [35] propose a combination of hardware and software components to control access to an embedded peripheral bus. They ensure availability for a multi-master I^2C bus in three ways: (*i*) only peripherals required by the currently executing application are allowed access to the bus, (*ii*) the currently executing application can only access the peripherals which it is allowed to access, and (*iii*) the length of any bus transaction is bounded. Their approach therefore protects against misbehaving applications that access unnecessary peripherals, as well as against misbehaving peripherals that do not adhere to the bus protocol. In short, Masti et al. rely on a trusted software layer to initialise the access rights and a dedicated hardware bus manager to enforce peripheral access control.

The Sancus MSP430 prototype uses a single-master SPI peripheral bus that is controlled through MMIO addresses.[2] The generic access control mechanism presented in this chapter could therefore be used to realise access control and availability guarantees for the SPI bus. To do so, a dedicated SM_{bus} should claim the relevant MMIO addresses in its private data section and implement an access control software layer on top, so that it mediated access to the SPI bus. Instead of file names, clients should supply a logical device identifier and SM_{bus} maintains an ACL of `(smID, permissions_flag)` per logical device. If access is allowed, SM_{bus} toggles the appropriate Slave Select line of the SPI bus to start the data transfer. Analogous to the approach of Masti et al. [35], such an SM_{bus} module ensures that clients can only access the peripherals they are allowed to. In contrast to Masti et al. however, SM_{bus}'s software layer cannot prevent misbehaving SPI peripherals from denying the availability of the bus by not adhering to the bus protocol. Such guarantees would indeed require hardware modifications to the SPI bus.

The next chapter shows how combining the above access control guarantees for shared system resources with a secure scheduling model results in strong availability guarantees for SMs, without loosing their hardware-enforced protection guarantees.

4.4.3 Limitations

This section acknowledges several limitations of the SM_{sfs} prototype. First, the limitations of the test set-up are discussed, thereafter the more fundamental issue of persistence and possible solutions.

[2] The technical details presented in this paragraph result from personal communication with Job Noorman.

Limitations of the Set-Up

A first limitation of the test set-up is that the Coffee case study back-end file system runs in unprotected mode. This should be no fundamental issue, since protecting the Coffee implementation should be feasible, as demonstrated by the initial inode-based elementary protected file system prototype of Sect. 4.2.1.

A second limitation concerns the protected flash driver. As explained in Sect. 3.1.3, Sancus' current program counter based MAL hardware circuits only allow a single contiguous private data section per SM. This implies that a module including a MMIO address range in its private data section, cannot at the same time store protected data. To realise end-to-end file system protection, one therefore needs a separate dedicated flash driver SM, exclusively communicating with SM_{sfs}. From a security perspective, there is no real issue here, but switching protection domains decreases the performance, as explained in Sect. 4.3.1. In a real-world set-up however, Sancus' combinational MAL circuits [40] that realise program counter based memory access could relatively easy be extended. The hardware logic could for example allow a single SM protection domain to consist of a contiguous protected address range to save private data and call stack, as well as a MMIO address range for exclusive access to a peripheral. While such an extension would increase the hardware costs by introducing additional registers and comparators per MAL circuit, it would also increase the overall flexibility of the system.

Limitations of the Non Persistent Approach

The SM_{sfs} prototype ensures confidentiality and integrity of logical files as long as it is up and running (which can be verified by the client), but does not persist these guarantees across reboots. Indeed, Sect. 3.1.5 explained the smIDs assigned to SMs by Sancus increase monotonically during a boot cycle. The assigned smID may therefore change when redeploying the same SM after rebooting the system. One could furthermore argue that extending SM_{sfs}'s file protection guarantees across reboots is non-trivial, as anything could happen between crashing of the node and successful redeployment of SM_{sfs}. In this respect, the protected file system does also not protect against physically removing and reading out the flash drive. This matches Sancus' attacker model, introduced in Sect. 3.1.1, which does not consider attackers with physical access to the hardware.

The existing non-persistent SM_{sfs} prototype should be considered as a way for SMs to extend their fixed sized private data section considerably, while at the same time offering flexible access control guarantees. In this respect, it could be an interesting future work direction to ensure the hardware automatically clears the flash drive on system boot – even before SM_{sfs} is deployed – to enforce the non-persistence of file system data.

The following briefly discusses several strategies to support persistence file protection and their downsides, which may or may not be problematic depending on the application scenarios and the desired security guarantees.

Trusted Loader A straightforward, yet undesirable solution would be to introduce a trusted SM_{init} module that "magically" initialises access control rules at boot time, after loading the SM_{sfs} module. Such a solution would reflect a static deployment scenario where the same SMs are always deployed on the same node and access the same files. To initialise file permissions, SM_{init} may proceed as follows. First, it loads all desired SMs and retrieves their smIDs using the `sancus_get_id` instruction. Thereafter, it loads the SM_{sfs} module and uses a special entry point to provide initial file permissions for the smIDs. The SM_{sfs} entry point should use the `sancus_verify_caller` instruction to make sure it has been called by SM_{init}. From then on, SM_{sfs} operates as usual and decides access using its internal file permission table.

From a security perspective the solution sketched above is clearly inferior, since it introduces a kernel-like trusted software layer. All clients SMs should indeed incorporate SM_{init} in their TCB for file protection. In this, the solution resembles Trustlite's Secure Loader [30] software entity, introduced in Sect. 3.2. In contrast to Trustlite, SMs still do not need to trust SM_{init}'s load process for their Sancus-provided security guarantees, as discussed in Sect. 3.1.3. SM_{init}'s load process becomes trusted however for their additional file protection guarantees, which is undesirable. Moreover, this solution neglects the possibility that an attacker may have tampered with the file system before SM_{sfs} has been re-enabled.

Long-Term SM Identity To exclude a kernel-like software entity from the TCB for file protection, SM_{sfs} should find another way of reliably authenticating modules over their lifetimes. Recall from Sect. 3.1.2 that the identity of an SM is based on (*i*) the content of its code section, and (*ii*) its layout (i.e. the load addresses of its code and data section). Using a hash of the code section alone is not a reliable option, since an attacker may deploy its own SM with exactly the same code section – which may or may not be problematic. Identifying a module through a hash of its layout plus code section allows different instantiations of the same module, but still does not protect against an attacker that deploys its own module at the address of his choice after rebooting the system.

The protected file system needs help from a persistent trusted third party to reliably identify a module across system reboots. The client SM's software provider could act as such a trusted third party. Section 3.1.6 indeed explained how Sancus can provide a confidential communication channel between an SM and its remote software provider. This channel could be used to safely transfer a key/capability to the client module. SM_{sfs} would then decide access based on this long-term capability, rather than the module's smID. To speed up subsequent accesses, the smID could of course be cached once the initial authentication through the capability is over.

The solution sketched above however still does not protect against an attacker that may have tampered with the file system before SM_{sfs} has been re-enabled.

File System Encryption The only way to protect against an attacker with access to the file system before SM_{sfs} has been re-enabled, would be to encrypt all data

on the flash disk with SM_{sfs}'s module-specific key. Encryption/decryption with this key is possible from within SM_{sfs}'s code section through the `sancus_wrap` and `sancus_unwrap` instructions, as discussed in Sect. 3.1.6. As evident from Eq. (3.3), SM_{sfs}'s hardware-generated key can be kept unchanged across system reboots by simply redeploying SM_{sfs} at the exact same load addresses.

Encrypting all data on the flash disk would however dramatically reduce performance, especially since all data is transferred safely through CPU registers on a byte-per-byte basis. To avoid encrypting every byte separately, SM_{sfs} would have to introduce caching in its private data section. This would transform the lightweight trusted access control software layer into a fully fledged file system of its own. Moreover, there would be little advantage over the situation where clients encrypt the data themselves before passing it to SM_{sfs} or even an unprotected file system.

4.4.4 Comparison with Other PMAs

While the protected file system prototype is implemented for the Sancus platform [40], the approach of its access control front-end is quite generic. That is, in a broader perspective the access control mechanism introduced in this chapter demonstrates the feasibility of supplementing the security properties offered by PMAs with SM-grained access control guarantees enforced by a protected software TCB. This section briefly discusses how SM_{sfs}'s approach can be ported to other PMAs. When doing so, it is important to distinguish between the protected shared memory use case and the protected shared flash storage use case.

Protected Shared Memory

The protected shared memory use case can be considered as an example of emulating flexible policies on top of Sancus' constrained hardware mechanisms. Indeed, as explained in Sect. 3.1.3, Sancus does not natively support complex policies such as dynamically allocating or sharing of protected memory. The Trustlite architecture [30], introduced in Sect. 3.2, does however provide a more flexible hardware mechanism through a configurable EA-MPU table. Trustlite natively supports protected shared memory between protection domains, but each shared memory region should be set up by the Secure Loader at boot time and requires one EA-MPU table entry per module that should be allowed access. Trustlite's flexibility is therefore limited by the number of entries in the EA-MPU table, which is defined at hardware synthesisation time.

Sancus' existing hardware protection model could also be extended to natively support some form of protected shared memory. The Fides [48] PMA for example provides a shared memory segment that belongs to the currently executing module. This allows a module to write data to the shared memory zone before transferring control to the next one, which automatically becomes the receiver of the data. While Fides relies on a trusted hypervisor to enforce memory access rights, such a shared memory zone could also be enforced in hardware, analogous to the existing

combinational MAL circuits that define SMs. While such a solution would drastically simplify basic sharing – and therefore also the performance of SM_{sfs} as discussed in Sect. 4.3.1 – it also illustrates some of the caveats of implementing more advanced policies in hardware. To start with, this solution still does not natively support complex forms of sharing between SMs. Think for example about sharing between three or more SMs, longer-duration sharing, etc. Moreover, in order to retain confidentiality and integrity, writing out the shared data and calling the next SM should happen in a single atomic operation. Realising such a guarantee can be non-trivial in a preemptive multithreading environment.

Protected Shared Flash Storage

The flash storage use case illustrates the more general case of controlling access to a shared system resource. In principle, SM_{sfs}'s generic front-end could be used to control access to *any* system resource that can be accessed through the memory address space – and thus encapsulated in its own protection domain. In this, the flash storage use case illustrates how SMs can supplement their PMA-provided security guarantees with access control guarantees for a shared system resource through a minimal and explicit software TCB. This chapter has identified the minimal set of mechanisms that should be provided by a PMA to support such secure resource sharing. They are (*i*) memory isolation, (*ii*) efficient caller authentication, and (*iii*) exclusive use of MMIO ranges.

The Trustlite [30] architecture, introduced in Sect. 3.2, provides all the required mechanisms. The secure resource sharing approach proposed in this chapter could therefore be ported relatively easy to this architecture. There is one caveat however: protected modules in Trustlite can only establish a mutually authenticated channel through a three way handshake protocol, as described in Sect. 3.1.5. In short, this implies that an SM_{sfs} module running on Trustlite cannot use a `sancus_get_caller_id`-like instruction to authenticate the client requesting the service. Instead, a client module should pass a secret nonce with every request and provide an entry point where SM_{sfs} can call back to ask for acknowledgement that the client has indeed requested a service from SM_{sfs} with that nonce. This would impose a serious performance overhead since switching protection domains comes at a cost, as discussed in Sect. 4.3.1. Trustlite's hardware logic could of course be extended to provide a `sancus_get_caller_id` alike or SM_{sfs} could agree on a secret session token with its client during the initial mutual authentication phase. An advantage of the Trustlite architecture on the other hand is its native support for multiple private sections per protection domain. This allows SM_{sfs} to incorporate the MMIO regions of the flash drive as well as another region for its private data and stack in its protected memory section, eliminating the need for a separate flash driver SM as introduced in Sect. 4.2.5.

Strackx et al. [49] propose to use a trusted software module, referred to as the *vault*, to securely offer persistent storage for client modules. The vault is similar to SM_{sfs} in that it stores secret data on behalf of its clients and guarantees to only return data to the module that initially saved it, but does not feature logical file

sharing between client modules. Strackx et al. [49] introduce the vault to be able to initialise the private data section of a module when it is (re)loaded. The vault therefore supports persistent storage across system reboots. To do so, they (*i*) encrypt and sign all data on the untrusted persistent storage device with a cryptographic key that is only known by the vault, (*ii*) base client authentication on the long-term identity (including a hash of the code section and layout information), and (*iii*) trust part of the software boot process to automatically load the vault and provide it with its secret key. Remark that these strategies and their downsides were discussed in Sect. 4.4.3.

4.5 Conclusion

This chapter presented a protected file system SM_{sfs} module that builds upon Sancus' existing hardware primitives to construct a software layer that realises logical file access control guarantees for client SMs. The trust relationship between SM_{sfs} and its clients differ significantly from a traditional omnipotent OS kernel for two major reasons: (*i*) clients can authenticate SM_{sfs} reliably, and (*ii*) clients only rely on SM_{sfs} for its dedicated file system task.

In a broader perspective, the prototype demonstrates the feasibility of supplementing the hardware-enforced security properties offered by PMAs with SM-grained access control guarantees for a common system resource through a lightweight protected software TCB. The access control approach discussed above is quite generic and can be implemented on other PMAs, or employed to control access to resources other than a file system. As such, this chapter has shown how a protected module could represent more than the unit of memory protection, but also the unit of secure resource sharing.

The resource sharing approach of this chapter presupposes that the resource can be encapsulated through the memory address space. Securely sharing of the more abstract CPU time resource is discussed in the next chapter.

Chapter 5

Secure Scheduling

This chapter presents a protected scheduler for the Sancus platform [40]. The scheduler is encapsulated in its own SM_{sched} protection domain and realises a form of cooperative multitasking where logical threads execute in the protected single-address-space. As such, this chapter implements a typical OS responsibility without introducing a privileged omnipotent kernel software layer. The protected scheduler supplements the hardware-enforced memory protection guarantees for SMs with availability guarantees for logical threads.

This chapter is organised as follows. Section 5.1 first reviews existing ideas and challenges for implementing multithreading in a single-address-space with fine-grained standalone protection domains. Thereafter, Sect. 5.2 introduces a multithreading scheme for the Sancus architecture and Sect. 5.3 presents the accompanying secure scheduler SM_{sched} implementation. Section 5.4 discusses the security and availability guarantees of the scheduler prototype and compares the approach to other PMAs. Section 5.5 finally concludes.

5.1 Threading in a Protected Single-Address-Space

A logical *thread* represents the unit of execution within an address space. A thread corresponds to a call stack representing the control flow and local variables. Multiple such flows of control may execute in the same address space, allowing shared data between them. Threads should not be confused with *processes* representing conventional OS entities, each with its private virtual address space. In a conventional multiple-address-space OS [46, 4] a process represents the unit of protection whereas a thread represent the unit of execution within such a protection domain.

To represent control flow in a single-address-space, one thus only needs the concept of logical threads, not processes. It is however not clear how these threads relate to any fine-grained protection domains in the shared address space. The following sections therefore review existing approaches to realise threading in a protected single-address-space to arrive at some basic notions for multitasking in a hardware-enforced PMA.

5.1.1 Opal Approach

The Opal [8, 9] single-address-space OS features logical threads executing in a shared address space with fine-grained protection domains. An Opal protection domain corresponds to an ACL with capabilities for memory segments. As opposed to a conventional OS thread that executes in a process's private address space, an Opal thread may run through multiple protection domains. At all time, an Opal thread is associated with one protection domain that defines its current memory access rights. In other words: "A protection domain is an execution context for threads, restricting their access to a specific set of segments at a particular instant in time." [9]. Threads may switch protection domains by invoking a system call to the Opal kernel. Such a system call continues the thread's execution at the predefined entry point of the new protection domain.

By decoupling the unit of execution from the unit of protection, Opal shows the feasibility of multiple memory protection contexts within a single thread's lifetime. This opposes to traditional private-address-space systems where a process can have multiple threads, but a thread cannot span multiple processes. Indeed, processes can only communicate indirectly by passing messages via Inter-Process Communication (IPC) techniques. In these systems, there is no notion of continuing execution in another protection domain.

5.1.2 Trustlite Approach

Trustlite [30], introduced in the Sect. 3.2, is a PMA that realises hardware-enforced protection domains, called *trustlets*, in a single-address-space. Trustlite's threading model regards a trustlet as a protected standalone task, "designed and believed to implement a particular security mechanism" [30]. A trustlet has a single internal protected call stack, and is therefore monothreaded. Interestingly, Trustlite offers a modified hardware interrupt engine that allows secure interruption of trustlets with a zero-software TCB. On interrupt and before jumping to the untrusted ISR, the hardware saves the program counter, stack pointer and CPU registers in the protected data section of the trustlet. The secure hardware interrupt engine thus ensures that even in the presence of interrupts, a trustlet solely depends on hardware for integrity and confidentiality of its internal state. Such a set up allows an untrusted preemptive scheduler to schedule trustlets in between normal unprotected tasks.

By considering a trustlet as a standalone task, Trustlite's threading model differs from that of Opal described above. Trustlite maps the trustlet as the unit of memory protection on the unit of scheduling, as with conventional OS processes. This seems to imply that calling a trustlet does not simply mean continuing the current thread's execution in another protection domain as in Opal. On the one hand, Trustlite indeed categorises calling a trustlet as message passing IPC. On the other hand, such a trustlet call in the single-address space simply comes down to "jumping to the respective entry points with arguments in CPU registers" [30].

An important question in a preemptive scheduling environment, is what a previously suspended trustlet should do when receiving a new request. Since a trustlet has

only one internal call stack, it cannot directly handle this request. In this respect, the Trustlite paper mentions that a trustlet "may simply queue the signal in a message buffer reserved in the trustlet data region" [30]. It remains unclear however what should be done when this internal buffer fills up.

5.1.3 Intel SGX Approach

Intel SGX [36, 27], introduced in Sect. 2.2.3, is a set of hardware extensions for high-end multiple-address-space computer architectures. SGX allows the creation of hardware-protected *enclaves* within an application's private virtual address space. Such an SGX enclave is a standalone hardware-protected module with a public interface and its own private call stack, heap and code section. Furthermore, SGX features a hardware exception engine that allows secure interruption of running enclaves by saving the state inside the protected section. SGX thus allows for an additional fine-grained layer of hardware-enforced memory protection on top of the coarse-grained virtual memory protection scheme managed by the untrusted OS.

SGX is not targeted at a single-address-space architecture, but offers enclave protection domains in the single private address space of a process. Consider an application with multiple threads, each wanting to use the enclave's services in the process's virtual address space. Since enclaves can be interrupted, this situation resembles that of Trustlite, described above. SGX however offers the concept of multithreaded enclaves. To do so, SGX keeps a Thread Control Structure (TCS) data structure per enclave, recording all thread-related meta data. Furthermore, SGX offers `EENTER`, `ERESUME`, `EEXIT` and `AEX` processor instructions [27] to respectively start a new enclave thread, resume an existing one and exit or interrupt the current one. When invoking `EENTER`, one must supply the address of a TCS inside the enclave to specify the logical enclave thread to be started. The enclave will subsequently check that the specified TCS is not busy yet. An enclave thread is considered busy from the moment it is started with `EENTER` until it is finished with `EEXIT`. A busy thread that was interrupted with `AEX` can be resumed with the `ERESUME` instruction that takes a pointer to a TCS as an argument.

SGX's threading model resembles that of Opal in the sense that conceptually the unit of execution – i.e. a thread inside a process's virtual address space – may run through multiple fine-grained protection domains. SGX however does not rely on an omnipotent kernel software layer to switch protection domain. Indeed, enclaves are (re-)entered via hardware instructions with a software-provided TCS thread identifier argument [27]. SGX's threading model differs from that of Trustlite since (*i*) protection domains can internally be multithreaded, and (*ii*) enclaves are not considered as separate tasks, rather as another protection context to continue the current thread's execution.

5.2 Threading and Control Flow in Sancus

From the above we can conclude that there are two major challenges to provide multithreading in a Sancus-like hardware-enforced PMA. First, an omnipotent kernel

software layer managing threads and their protection domains is inherently impossible and undesirable as it would invalidate the zero-software TCB concept. SMs should indeed exclusively rely on the hardware to retain confidentiality and integrity of their internal state. Independent from these guarantees, SMs might trust a scheduler for additional CPU availability guarantees. Second, the unit of logical threading in a single-address-space does not coincide with the SM as the unit of protection.

Recall from Sect. 3.1.7 that Sancus allows calling an SM's entry function from unprotected code or from another module. Sancus thus already has the concept of a single logical control flow thread running through multiple SM protection domains. Assembly code stubs inserted at compile time take care of restoring the internal state when switching hardware protecting domains. The essence of the multitasking model presented in this chapter is to extend this monothreaded scheme to securely allow multiple such control flows to exist simultaneously.

This section covers some general concepts that must be discussed before the implementation of the protected SM_{sched} module. The explanation is organised as follows. Section 5.2.1 first provides an helicopter view of the threading model and division of responsibilities between the scheduler and the individual participating SMs. Section 5.2.2 thereafter resolves some remaining issues regarding control flow integrity in the existing SM calling mechanism.

5.2.1 Threading Model Overview

As in the existing control flow model, Sancus threads will be allowed to run through multiple SMs during their lifetimes. The scheduler only keeps track of logical threads, each with their current SM protection domain where execution should be continued. In terms of the threading models discussed in Sect. 5.1, this model resembles that of Opal [9] in that SMs are regarded as an execution context for logical threads, and contrasts that of Trustlite [30] since SMs are not considered as standalone schedulable units.

In contrast to a conventional scheduler [46, 50], a scheduler for Sancus cannot save the thread's current state (i.e. CPU registers, stack pointer and program counter). The thread's internal state is in fact distributed over all the SMs that are part of it. Recall from Sect. 3.1.7 that SMs indeed have their own private call stack and agree on a protocol to call and return to each other. Sancus' existing control flow model thus already supports a single implicit logical thread that is not represented by a single call stack – as in traditional systems – but by a private call stack per protection domain. Dedicated code stubs executed when entering or exiting an SM are responsible for saving and restoring private stacks. Logical control flow is thus jointly realised by the participating SMs, whereas memory isolation and protection domain switching is guaranteed by the Sancus hardware. As will be shown in Sect. 5.2.2 however, the existing `sm_entry` procedure is not sufficient to enforce control flow integrity and must first be extended accordingly.

The scheduler is only concerned with scheduling and relies on the cooperation of individual SMs in three ways. First, due to the lack of a secure hardware interrupt engine, SMs are expected to voluntarily hand over control to the scheduler from

time to time. Second, SMs should guard the entry of their protection domains to ensure control flow integrity. Third, SMs that are part of multiple logical threads are responsible themselves to internally work for one thread at a time. They should not mix the contexts of different logical threads on the same private call stack. SMs should therefore be made *threading-aware*. How this is realised is explained in detail in Sect. 5.3.3.

It is in the best interest of an SM to guard the entry of its protection domain to ensure the integrity of its internal control flow. From a security perspective, a non-cooperating SM can only harm itself or invalidate the control flow of the logical thread that it is part of. The latter will be detected by other participating SMs and cause the killing of the thread (as explained further on).

5.2.2 Sancus Control Flow Integrity

As explained in Sect. 3.1.7, SMs consist of a single physical entry point that executes a short assembly code stub, referred to as `sm_entry`, to switch the private call stack and to allow calling multiple logical entry functions. To do so, the Sancus compiler assigns a logical eIDX identifier to each entry function and generates the `sm_entry` stubs. Next, the compiler replaces all calls to such functions with a jump to the physical entry point, providing the eIDX and return address via fixed CPU registers.

Nothing stops an attacker from calling the physical SM entry point himself however, providing arbitrary values in the CPU registers. The `sm_entry` assembly stub should therefore properly check its arguments. If any argument value is clearly wrong, the caller is disobeying the SM entry protocol and should not be allowed to enter. The following explains the different kinds of illegal arguments, how they can be detected and how to respond to such an entry violation.

Remark that the use of the term "control flow integrity" in this chapter differs from the traditional concept. Abadi et al. [1] introduce control flow integrity as a mitigation technique that inserts runtime code checks to ensure execution adheres to a valid path in a precomputed control flow graph. While the techniques introduced in this section rely on runtime checks at the boundaries of protected modules, they do not use a control flow graph.

Logical Function Index Out of Bounds

A straightforward entry protocol violation is to provide an illegal eIDX. Since this is an index in a fixed-sized lookup table, one can easily verify it is within bounds, as has already been done in the existing `sm_entry` code stub.

Arbitrary Return Address

The caller is expected to provide the return address where execution should be continued when the module being called has finished. Since the existing `sm_entry` code stub does not check this address, an attacker is able to execute arbitrary code within the module being called. Indeed, the last instruction that is executed in the callee SM is an unconditional jump to the provided return address. Since this

instruction is executed from within the module, jumping to an internal address is allowed by the hardware-enforced program counter based access control rules.

As an example of such an attack, consider the secure file system SM_{sfs} module from Chapter 4. Recall that the module consisted of a public front-end access control layer, controlling access to a private back-end API. An attacker might bypass the access control layer as follows. First, he figures out the address of the private back-end helper function that he needs and moves this address in the corresponding CPU register. Next, he jumps to SM_{sfs}'s physical entry point, providing a valid eIDX of an arbitrary logical entry function via the corresponding CPU register. The front-end entry function being called will perform the necessary access control, deny access to the back-end and jump to the provided return address. By providing the address of a back-end function, the attacker thus ensures execution is continued as if the front-end allowed access. Two things should be noted here. First – in the absence of buffer overflows and other well-known [18] low-level security attacks – the attacker has no control whatsoever over the callee SM's internal call stack. The attacker can only jump to arbitrary code once, by providing the return address to the `sm_entry` stub. This implies that he cannot mount a return oriented programming attack [45]. Second, the attacker has no direct control over the argument CPU registers on the moment of the return call. They will either be cleared or contain a return value from the entry function.

From the above, it should be clear that it is in the best interest of an SM to verify the provided return address. To do so, the callee can employ the existing Sancus hardware primitives, introduced in Sect. 3.1.5, as follows. The SM being called first retrieves the caller's smID with the `sancus_get_caller_id` instruction. Thereafter, the callee uses the `sancus_get_id` instruction to retrieve the smID of the SM that owns the return address. The callee then simply compares these smIDs to make sure the provided return address is indeed owned by the caller.

Since every protected module should have its own call stack, providing a continuation address when calling a module is a general technique that is used by many PMAs. In this respect, Agten et al. [2] present a compilation scheme that preserves the security guarantees of a program expressed in a Java-like language when compiled to a low-level PMA featuring program counter based access control. Their compilation process inserts runtime checks at the boundaries of protected modules, analogous to Sancus' compiler-generated stubs described in Sect. 3.1.7. Agten et al. [2] preserve integrity of the control flow by verifying that an externally supplied address always lies outside the module's memory bounds. Remark that their countermeasure is analogous to the one of the previous paragraph, but that the `sancus_get_caller_id` instruction allows for a stronger assurance. That is, the countermeasure introduced in this section ensures that the return address is owned by the caller, whereas the approach of Agten et al. [2] only ensures that the return address is not owned by the callee. From a security perspective both options are sufficient, but the latter allows to "return" into a module that has no open return entry point. This scenario is discussed in the next section.

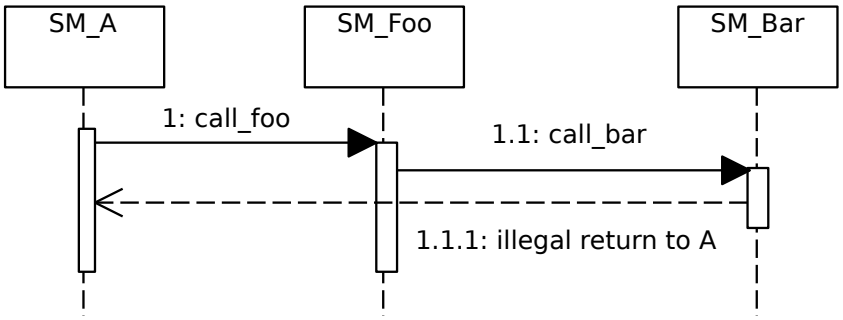

FIGURE 5.1: SM_{bar} bypasses SM_{foo} by abusing an open entry point at SM_a

Abuse of Open Return Entry Points

As explained in Sect. 3.1.7, the SM entry protocol uses a special eIDX identifier to re-enter an SM after calling an external function. The existing `sm_entry` assembly stub does however not verify that (*i*) the SM being re-entered has indeed called an external function, and (*ii*) the calling SM is the one that was called.

An attacker might thus violate the entry protocol by calling an arbitrary SM with the special return eIDX. This will result in unpredictable behaviour, as it is not even sure the callee SM is waiting for such a return. More subtle attacks that violate control flow integrity are also possible. Figure 5.1 for example depicts a situation where a malicious SM_{bar} module abuses an open entry point at SM_a, bypassing the intermediate SM_{foo} module. SM_a is unaware of this bypass and believes SM_{foo} has returned. The SM_{foo} module is left with an open entry point, waiting for SM_{bar} to return.

To detect such control flow integrity violations, an SM should be able to tell on every re-entry (*i*) whether it has called an external function, and (*ii*) the smID of the last called module. The first requirement can be verified by inspecting whether the module's private call stack is empty. Indeed, as explained in Sect. 3.1.7, the compiler automatically inserts an `sm_exit` assembly code stub on every external function call to store all registers and save the private stack pointer before exiting the module. Simply verifying whether the stack pointer points to the stack's fixed base address on re-entering, therefore suffices to verify the SM is indeed waiting for a return. The second requirement can be fulfilled by employing existing Sancus hardware instructions as follows. First, the calling SM's `sm_exit` stub is modified to save the smID of the module it is about to call on top of its private stack. This smID can be retrieved by the `sancus_get_id` instruction with the callee's address as an argument.

When re-entering an SM with the special return eIDX, the callee first checks whether the stack is empty. If so, the caller clearly violates the entry protocol. If not, the smID of the SM that is expected to return is saved on top of the stack. The callee can thus pop this smID and compare it with the result of the `sancus_get_caller_id` instruction to ensure control flow integrity.

The secure compilation scheme proposed by Agten et al. [2] includes a coun-

termeasure that prevents "returning" into a module that has no open entry point. To do so, their compiler initialises a module's private call stack with the address of an internal function that halts the system. Since the internal return address is popped from the stack, execution is halted on an illegal re-entry. Observe that this countermeasure is analogous to checking whether the private call stack is empty when re-entering a module. Agten et al. [2] presuppose a single protected module for their secure compilation scheme. They therefore do not have to consider the more subtle attack where control flow integrity is violated by bypassing an intermediate module. They do however acknowledge that "new attack vectors might exist due to the increased complexity of multiple interacting modules" and that "new compiler measures would have to be installed, to protect against these new attacks" [2]. The countermeasure presented in this section can well be considered as an example of such a new compiler measure, relying on the `sancus_get_caller_id` instruction

Reporting Entry Violation to the Scheduler

The above sections explained how SMs should guard the entry of their protection domain by validating the provided arguments. They did not yet discuss how SMs should react to these entry violations. Clearly, a caller disobeying the entry protocol should not be allowed to enter and execution should stop. In a non-scheduling environment, the callee may simply go into an infinite loop. In a multithreaded environment, it would be desirable to halt the currently executing thread, without affecting others. To do so, the SM that denies access should keep its internal state consistent and inform the scheduler of the entry violation. The scheduler in its turn should forget about the currently executing thread, since this thread has halted at the reporting SM.

To be able to securely call the scheduler from within an `sm_entry` assembly code stub, one needs SM_{sched}'s (*i*) load address, (*ii*) smID, and (*iii*) desired eIDX entry function identifier. Since these values are only known when the SM_{sched} module has successfully been loaded, SMs should be informed of them at run time. For this purpose, all SMs keep fixed locations within their private data sections to store this information. These locations are given symbolic names by the linker script, so that they can be initialised from a standard C entry function. This function simply fills in the corresponding symbolic values and retrieves the smID using the `sancus_verify_address` instruction and a MAC of SM_{sched}, as explained in Sect. 3.1.4. The main function calls this initialisation function for each of the participating SMs, after SM_{sched} has been enabled and before starting the scheduling. Appendix A includes a simple C code example that illustrates this initialisation process in a static deployment scenario (where the participating SMs are statically linked against SM_{sched} before being sent to the node).

When provided with the above values, an SM detecting an entry violation will first verify whether the scheduler is still loaded correctly at the expected address. This can be accomplished by passing the stored scheduler address to the `sancus_get_id` instruction and comparing the result with the stored smID. If the scheduler is not present, something has clearly gone wrong and the detecting SM has no choice but

to go into an infinite loop. In the other case – when SM_{sched} has successfully been verified – the detecting SM can simply jump to its entry point with the desired logical entry function eIDX in the correct CPU register.

5.3 Implementation of a Protected Scheduler

This section presents the implementation of the protected scheduler module. The scheduler is encapsulated in its own SM_{sched} protection domain and realises cooperative multitasking where multiple logical threads share a single CPU and protected single-address-space.

The explanation is organised as follows. Section 5.3.1 presents the interface of the SM_{sched} module and Sect. 5.3.2 discusses how the scheduler internally represents logical threads. Thereafter, Sect. 5.3.3 presents the idea of *threading-aware* protection domains that ensure internal consistency in a multithreading environment. Section 5.3.4 finally explains how the scheduler manages to interweave the execution of multiple logical flows of control.

5.3.1 Scheduler Interface

Table 5.1 summarises the interface of the protected SM_{sched} module. The functions fall down in three categories. A first kind of functions allow the main thread to set up and start the scheduler. Functions of the second category are used by individual SMs to participate in the scheduling process. The third function category consists of private helper functions that make the actual scheduling decisions.

TABLE 5.1: Overview of the interface offered by SM_{sched}. The "main" rows contain functions to set up the scheduler. The "SM" rows correspond to functions used by participating SMs. The "SM_{sched}" rows are private helper functions.

Used By	Function	Description
main	`register_thread_portal`	Provide entry function of new thread.
	`start_scheduling`	Run previously registered threads.
SM	`yield`	Hand over control to scheduler.
	`report_entry_violation`	Control flow integrity has been violated.
	`get_cur_thr_id`	Return the current thread id.
SM_{sched}	`finish_get_next`	Current thread is done, return next one.
	`yield_get_next`	Suspend current thread, return next one.
	`kill_get_next`	Kill current thread, return next one.

Recall that logical threads might run through multiple protection domains. The first of these, the SM where the thread started, is referred to as its *portal*. The main function can use the `register_thread_portal` function to provide the scheduler with such a new logical thread, to be started later. This function expects the portal SM's address, smID and logical eIDX entry function identifier to start the thread.

When all logical thread portals are registered this way, the main function calls the `start_scheduling` function. The scheduler then runs all registered threads according to its internal scheduling policy and returns to the main function when all registered threads have finished.

As already mentioned above, individual SMs are expected to cooperate in the scheduling process in three ways. First, by calling the `yield` function they voluntarily hand over control to the scheduler, which might decide to run another logical thread and continue this one at a later time. A second way in which SMs participate in the scheduling process, is by reporting an entry protocol violation, as introduced in Sect. 5.2.2. This can be accomplished with the `report_entry_violation` function that kills the currently executing logical thread. Finally, the scheduler offers a `get_cur_thr_id` function that returns a unique identifier for the currently executing thread. This will be used in Sect. 5.3.3 to realise threading-aware protection domains that internally separate the call stacks of different logical threads.

The last category of scheduler functions consists of private helper functions that can only be called from within the SM_{sched} module. These functions are used to decide which logical thread to run next and thus make up the scheduling policy. Section 5.3.4 explains them in more detail.

5.3.2 Scheduler Internals

The following describes how SM_{sched} internally keeps track of logical threads and their currently executing SM protection domain.

Logical Thread States

Figure 5.2 depicts the different states that a logical thread might be in during its lifetime and the possible transitions between them. All threads start in the "registered" state, after they are created through the `register_thread_portal` function. The associated protection domain for a thread in this state is always the corresponding portal SM. When the scheduler select the thread and starts it through its portal, the thread is said to be "running". Since the prototype features a single embedded CPU, there is always at most one logical thread in the "running" state. When the currently executing SM calls the `yield` function, the current thread is suspended and moves to the "ready" state. The associated protection domain in this state is the SM that just yielded, which can be any SM that is part of the logical thread. At a later time, the scheduler might decide to continue the thread, which is then again said to be "running". To continue a logical thread's execution, the scheduler simply jumps to the address of the SM that called the `yield` function with the special return EIDX in the corresponding CPU register. From the point of view of the SM, it seems as if the scheduler just returned from the `yield` call and nothing happened in between. When the portal SM finally returns to the scheduler, the current thread is considered "finished". Recall from Sect. 5.2.2 that an SM may call the `report_entry_violation` function to indicate it has been entered incorrectly.

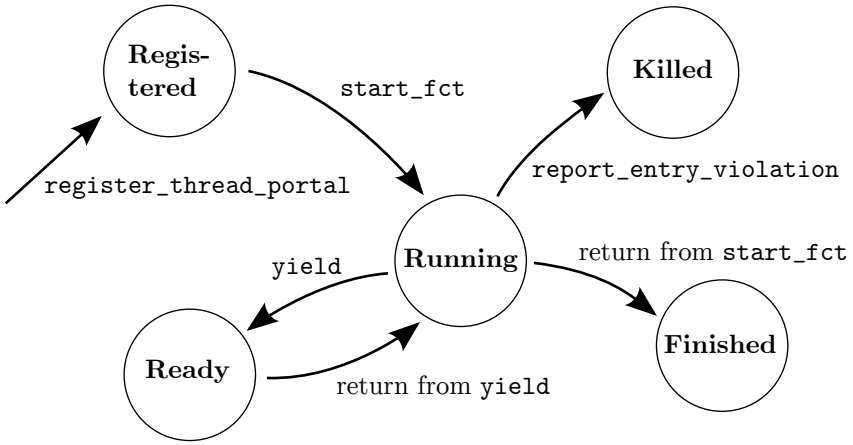

FIGURE 5.2: Logical thread states and their transitions as recorded by the scheduler

When this happens, the scheduler forgets about the currently executing thread and it is said to be "killed".

Data Structures

SM_{sched} stores all internal data structures in its protected data section. The scheduler maintains a pointer to the currently executing thread and a ready queue with "registered" and "ready" threads. A logical thread is represented by a *thread control block* that contains a unique thrID identifier and information on its current SM protection domain. More specifically, the implementation keeps track of the associated SM's entry address and logical eIDX entry point. For "registered" threads the eIDX corresponds to the logical entry function of the portal SM to start the thread. For "ready" threads, the eIDX is always the special return entry index. This allows the implementation to continue execution of a logical thread by simply jumping to the corresponding entry address, with the eIDX in the agreed CPU register.

Since Sancus' hardware logic imposes a fixed-sized private data section, SM_{sched} should fulfil its own dynamic protected memory requirements. Analogous to the protected file system implementation of Sect. 4.2.3, SM_{sched} pre-allocates a fixed number of thread control blocks at compile time and maintains them in a free list at run time. This implies a configurable maximum number of logical threads that can be managed simultaneously by the SM_{sched} prototype.

5.3.3 Threading-Aware Protection Domains

Different logical threads may use the services of the same SM. Consider for example the protected file system module SM_{sfs} from Chapter 4. If SM_{sfs} is executing for logical thread A and yields to the scheduler, another thread B might want to use its services. In this case, SM_{sfs} should however not mix the execution contexts of different logical threads on the same private call stack. Imagine for example that

SM_{sfs} simply pushes the call context of thread B on top of that of thread A. If the file system then yields to the scheduler once more, thread B will be continued instead of thread A since it is on top of the call stack. To ensure control flow integrity in a multithreading environment, SMs should therefore be made *threading-aware*.

As explained in Sect. 5.1.3, Intel's SGX [36] architecture features multithreaded enclave protection domains. SGX enclaves also separate the local execution contexts of these logical threads. To do so, SGX enclaves internally store the thread's metadata in a TCS data structure and demand the caller to specify the address of such a TCS as an argument when entering the enclave. In the case of the Sancus architecture with a single embedded CPU however, there is always at most one logical thread running. The logical thread should therefore not be specified explicitly when entering an SM, as this will always be the "currently executing thread". The SM_{sched} module therefore provides participating SMs with the possibility to retrieve a unique thrID identifier for the currently executing thread via the `get_cur_thr_id` function.

The key idea to realise threading-aware SMs is to retrieve the current thrID as part of the SM entry procedure. The thrID, uniquely identifying the current logical thread, then also defines which internal private call stack should be used to save execution context in the current protection domain. The prototype implementation only features a single call stack per SM, but due to the generic thrID scheme, SMs can in principle store execution context for more logical threads on multiple internal call stacks. In the case of a single private call stack, SMs should be executing for at most one logical thread at all times. This is referred to as *internal monothreading*.

To ensure internal monothreading, the `sm_entry` assembly code stub from Sect. 5.2.2 should be extended once more. Participating SMs store the unique thrID of the logical thread that they are executing for (if any) at a fixed location in their private data section. Recall that a participating SM already stores the scheduler's address and smID. To be able to securely retrieve the thrID of the currently executing logical thread, it thus also needs to store the logical eIDX identifier of the `get_cur_thr_id` function. Provided with the internal thrID and the thrID of the currently executing thread, an SM being entered first checks whether its single private call stack is empty. If so, the module was not executing for a logical thread before it was entered. The module can thus safely be entered and its internal thrID is replaced with the current one. If not, the SM being entered is currently already executing for a logical thread and can only safely be entered when its internal thrID matches the current one. If this is not the case, the callee SM is currently storing control flow context for a partially completed logical thread that is different from the currently executing one. As the callee has no internal call stacks left, it cannot safely allow entrance into its protection domain. In this case, the callee's `sm_entry` code stub returns a magic value in an agreed caller-saved CPU register to indicate that it is currently busy. The `sm_entry` assembly code stub of the calling SM should therefore check the agreed CPU register for the magic busy value on every re-entry from an SM call. If the callee indicated that it is busy, the caller should retry the call at a later time. In this case, the caller's `sm_entry` stub saves state on its internal call stack and calls the scheduler's `yield` function to suspend the execution of the current logical thread. When the caller is continued at a later time, its `sm_entry`

stub will automatically retry the call.

From a C programmer's perspective, the above concept of internal monothreading is completely hidden by the compiler-generated `sm_entry` assembly code stubs. The same goes for the control flow integrity checks from Sect. 5.2.2. The Sancus compiler thus always generates cooperating SMs that ensure internal consistency. Remark that it is in the best interest of an SM to guard the entry of its protection domain and ensure internal consistency. A hand-crafted malicious module can only harm itself or the overall control flow integrity of the logical thread that it is part of.

5.3.4 Logical Thread Switching

As mentioned above, the currently executing thread might return from its portal entry function to indicate completion, or call the `yield` function to suspend itself. Moreover, an SM might call the `report_entry_violation` function at all time to indicate the current thread should be killed. To make this all work, SM_{sched} employs a custom `sched_entry` assembly code stub that intercepts these calls and translates them to the private C helper functions from Table 5.1. In essence, these helper functions make the actual scheduling decisions. They mark the current logical thread accordingly and return the information needed to continue the next one via agreed CPU registers to the `sched_entry` stub. More specifically, to start or continue a logical thread from assembly code, the stub needs (*i*) the entry address of its portal or currently executing SM, and (*ii*) the eIDX that identifies the logical entry point in this module. Since this information is stored in the thread control blocks that form the ready queue, the C implementation may simply return the required information to the the assembly stub after selecting the next logical thread to run.

Note that a dedicated `sched_entry` assembly stub is needed since SM_{sched} cannot build up local stack context when switching logical threads. The call stack should indeed not grow so that when all threads have finished, SM_{sched} can return to the main function that initially called `start_scheduling`.

Appendix A includes the C source code and sequence diagram of a minimal multithreaded program to illustrate logical thread switching behaviour, as well as the threading-aware protection domains from the previous section.

Finishing the Current Thread

When the portal function that was initially called to start the current thread returns to SM_{sched}, the thread is considered finished. The `sched_entry` stub therefore intercepts calls that specify the special return eIDX to call the private `finish_get_next` C helper function. This function marks the current thread as finished (i.e. forgets about it) and chooses the next one from the ready queue. The SM_{sched} prototype schedules logical threads according to a simple first-in, first-out circular policy, but in principle any scheduling policy (e.g. priority based) can be implemented here. When the scheduling policy has chosen a thread control block from the ready queue, the function returns the corresponding eIDX and SM address to the `sched_entry` assembly stub. If there are no logical threads left in the ready queue, the function

returns a special value. The `sched_entry` stub now simply checks whether the function returned information to start the next thread. If so, the stub moves the eIDX in the correct register and jumps to the provided SM's entry address. If not, the stub performs a normal return to the main function that called `start_scheduling`.

Remark that if the `sched_entry` stub would process all returns in this way, the scheduler won't be allowed to call any external function that is not the (re-)entry point of a logical thread. The SM_{sched} implementation however uses unprotected library functions such as `printf` for debugging purposes. The `sched_entry` stub therefore uses the `sancus_get_caller_id` instruction to retrieve the smID of the calling protection domain, and allows normal re-entry from the unprotected domain. This implies that in the prototype implementation, the unprotected domain cannot act as the portal of a logical thread. Of course more flexible solutions are possible. An internal boolean flag could for example be used to indicate whether or not a logical thread is currently running.

Suspending the Current Thread

The currently executing logical thread may voluntarily suspend itself by calling the `yield` function. From the point of view of the calling SM this is an ordinary function and it continues normally when this call returns. To make sure execution is indeed continued when restarting the suspended thread, SM_{sched} should update the current thread control block with the information of the yielded SM. The scheduler already knows the eIDX as it is the special return entry index and it can retrieve the smID via the `sancus_get_caller_id` instruction. As with a normal function call, the caller's SM entry address is provided as the return address for the call via an agreed CPU register. This will be the address where execution is continued when resuming the current logical thread. It is therefore essential that the SM_{sched} implementation verifies this address is indeed owned by the caller with the `sancus_get_id` instruction, as explained in Sect. 5.2.2. If this verification fails, the current thread is disobeying the entry protocol and cannot be continued. The implementation then simply forgets about it, as explained further on.

Analogous to the return entry point interception discussed above, the `sched_entry` stub intercepts all `yield` entries to call the private `yield_get_next` helper function. This function updates the current thread control block with the calling SM's information, as discussed above, and appends the thread control block to the ready queue. Thereafter, the C implementation selects the next logical thread to run and returns the necessary information to continue the thread's execution to the `sched_entry` stub.

Killing the Current Thread

As explained in Sect. 5.2.2, additional integrity checks are performed when entering an SM and a module may use the `report_entry_violation` function to signal it has been entered incorrectly. Analogous to the above, the `sched_entry` stub intercepts these calls and translates them to a private `kill_get_next` call. This C helper

function makes the scheduling decision and returns information to continue the next logical thread, if any.

Since the integrity of the control flow of the current thread has been violated, it cannot be continued any more. Recall that a logical Sancus thread consists of the collection of all private call stacks of its participating modules. To actually *kill* a logical thread system-wide, all these call stacks should be cleared. Since SM_{sched} is not privileged in any way – as opposed to a traditional OS kernel – it cannot kill logical threads this way. The implementation therefore simply removes the current thread control block, so that it will not be assigned CPU time any more.

Terminating logical threads in this way has two implications. First, any currently executing SM (including the unprotected domain) is allowed to call the `report_entry_violation` function. While this has obvious availability implications, it also conforms to the model where a caller verifies and trusts a callee. Indeed, when calling an SM correctly, one expects that it accepts the call and returns eventually. A malicious module might on the contrary report an alleged violation to the scheduler, resulting in the termination of the currently executing thread. The key thing to note here however, is that this can *only* be done by modules that are explicitly called by some module in the logical thread and thus also form part of the logical control flow. At all times times only the currently executing thread can be killed. This implies that a logical thread A can never cause the killing of a logical thread B. By carefully deciding which modules to call and by guarding their entry, one can therefore construct a secure logical control flow that will always be allowed to run, independent from any other threads being scheduled.

A second consequence of the possibility to kill logical threads in the scheduler concerns the individual participating SMs. Recall from Sect. 5.3.3 that SMs are threading-aware and make sure that their internal private call stack only builds up context belonging to the same logical thread. When abruptly terminating the current thread, some of its participating SMs will be left with a partial call stack. Since the SM_{sched} implementation makes sure it never re-uses thrIDs, the participating modules will never continue the control flow represented by these partial call stacks. While this is indeed desirable from a control flow integrity point of view, it may also result in a deadlock. Figure 5.3 depicts such a situation where the gray logical thread consists of an SM_{foo} module that calls an SM_{bar} module which illegally calls SM_a. This last module detects the entry violation and informs the scheduler, which forgets about the gray thread and start the white one. The previously mentioned SM_{foo} module is however unaware the gray thread has been killed and therefore indicates that it is busy when the white thread wants to use its services. The essence of the deadlock is that SM_{foo} keeps a private call stack that waits for SM_{bar} to return, which will never happen.

The current prototype does not recover from such a deadlock, but control flow integrity will never be jeopardised. In the above example, SM_a will continuously retry to call SM_{foo} that keeps indicating it is busy. Avoiding the deadlocked situation is certainly possible, but requires extending the threading-aware protection domain approach from Sect. 5.3.3. Consider for example changing the definition of the `get_cur_thr_id` function so that it takes the locally saved thrID of the caller as an

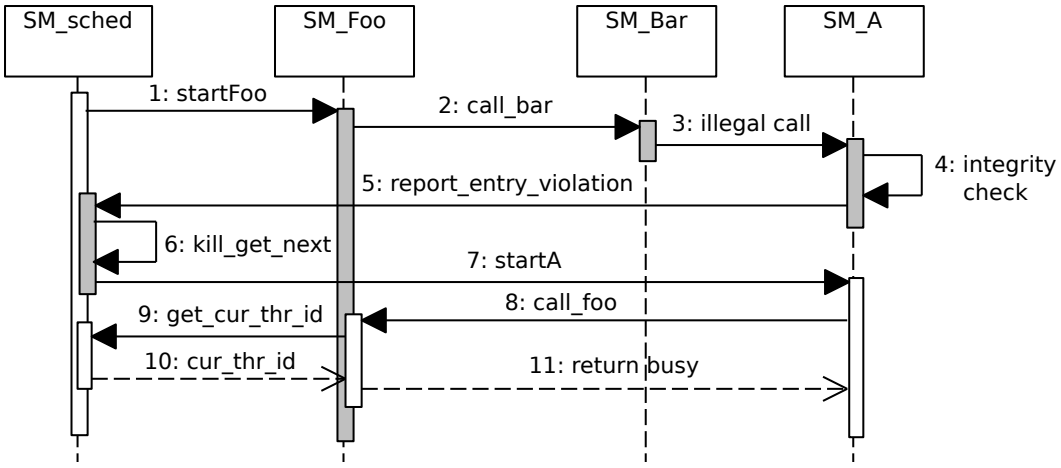

Figure 5.3: Multithreading scenario resulting in a deadlock. The gray thread is killed, but the participating SM_{foo} module is unaware of this and refuses subsequent calls from other threads because it still has an open call stack for the gray thread.

argument. SM_{sched} could then simply maintain a list of killed logical thrIDs and check whether the calling SM provided one of them. If so, this implies it was part of and still waits for a logical thread that is now killed. In such a case, the scheduler may return a special thrID to indicate that the calling SM is allowed to clear its entire current private call stack to participate in another logical thread.

5.4 Discussion

This section discusses the guarantees and limitations of the above threading model and SM_{sched} prototype. The approach is also briefly compared to that of other PMAs.

5.4.1 Security Guarantees

The key thing to note from a security perspective, is that the SM_{sched} prototype is not privileged in any way. In contrast to a conventional OS scheduler, SM_{sched} only encapsulates the scheduling policy. It cannot and does not save the execution state (i.e. program counter, stack pointer, CPU registers) of suspended tasks. Recall that a logical control flow thread indeed corresponds to all the private call stacks of the SMs that are part of it. Extending the `sm_entry` code stubs with integrity checks further ensured that SMs rely exclusively on the PMA hardware and their private implementation for the confidentiality and integrity of their internal control flow and data. Participating SMs should only trust the scheduler to be guaranteed CPU time. They should not incorporate SM_{sched} or any other module in their TCB – as opposed to a traditional omnipotent OS kernel.

Participating SMs rely on the thrIDs provided by SM_{sched} to properly separate their internal call stacks. A misbehaving scheduler that returns false thrIDs can only cause stack frames belonging to different logical threads to be intermixed on the same private call stack. While this makes it impossible to state which thread is currently executing, control flow integrity will never be jeopardised. To see why, recall from Sect. 5.2.2 that the smID of the callee is pushed on the private call stack of the caller before making the actual call. Consider a caller that builds up execution context for another logical thread on top of its original call stack, due to a misbehaving scheduler that returns an identical thrID. If the scheduler now continues the callee that returns to the caller, an entry violation will be detected, since the callee's smID is no longer on top of the stack.

From the point of view of a participating module, yielding to the scheduler is a normal external function call and "continuing" proceeds as an ordinary return entry call. The control flow integrity entry guards from Sect. 5.2.2 that prevented an SM from abusing an open entry point or "returning" into a module that has no open entry point, therefore also enforce that SM_{sched} can only continue SMs that have previously yielded.

The above separation of concerns between the scheduler and the participating modules furthermore allows for optimal flexibility. One could simply plug in another SM_{sched} implementation to realise a different scheduling policy. Moreover, as hinted in Sect. 5.2.2, SMs can choose for themselves how many internal logical threads they want to support by allocating multiple private call stacks. Multithreaded protection domains might be useful for SMs that offer a service towards others, e.g. the protected file system from Chapter 4.

5.4.2 Availability Guarantees

In terms of availability, the current prototype cannot make strong guarantees, due to the lack of a secure hardware interrupt engine for the Sancus [40] platform. This section therefore first discusses the availability guarantees that can be enforced by the current prototype and thereafter elaborates on the requirements and implications of extending the threading model with a secure hardware interrupt engine.

Cooperative Multithreading

The current prototype relies on the cooperation of individual SMs to voluntarily transfer control to the scheduler. That is, logical threads are ensured processing time on the condition that participating modules always `yield` at some time. Such a cooperative multitasking scheme already allows for useful applications. Earlier versions of TinyOS for example did not support a preemptive multitasking model [20, 29]. Without preemption however, a malicious or buggy module can easily launch a Denial-of-Service (DoS) attack by for example executing in an infinite loop.

Section 5.3.4 discussed how the current thread may be killed through the `report_entry_violation` function. It was argued that a logical thread A can never cause the killing of another thread B and that the associated availability im-

plications correspond to a model where a caller verifies and trusts a callee. This only holds however for threads that do not execute in the unprotected domain. To see why, consider a logical thread A that is executing in the unprotected domain and yields to SM_{sched}, which continues another thread B. Since (part of) A's execution context is stored on an unprotected call stack, thread B can easily modify the unprotected control flow for thread A. Secure threads in the current prototype should therefore never call unprotected code (e.g. `libc` functions).

An advantage of the Trustlite [30] architecture over the initial Sancus [40] architecture is that the former can handle memory access violations by an untrusted OS scheduler. Recall from Sect. 3.1.3 that the current Sancus hardware generates a non-maskable interrupt and vectors to the predefined ISR on memory access violation. The SM_{sched} prototype presented in this chapter could therefore relatively easy be extended to securely handle memory access violations as follows.[1] First, a helper module SM_{vector} is created with a text section that contains the start address of SM_{sched} and an empty data section. Next, SM_{vector}'s text section is wrapped around the memory access violation entry in the interrupt vector table, which is at a fixed location in the TI MSP430's memory. This prohibits an attacker from changing the ISR address where execution will be continued after a memory access violation has been detected. Finally, SM_{sched}'s `sm_entry` stub has to be modified so that it recognises an entry resulting from a memory access violation. This can easily be accomplished since the `sancus_get_caller_id` instruction returns a special value when the previously executing module was interrupted. SM_{sched} can now kill the currently executing logical thread that attempted the memory access violation, as explained in Sect. 5.3.4. Such a strategy thus allows that the Sancus architecture not only enforces memory access rights, but also handles them gracefully – without harming uninvolved SMs running on the same node.

Preemptive Multithreading

Section 3.3.2 compared Sancus' hardware primitives to that of a minimalist microkernel [33] and concluded that only a notion of threading seems to be missing. The work presented in this chapter has shown that Sancus' knowledge of the previously executing module is sufficiently strong to realise control flow integrity, as discussed in Sect. 5.2.2. To ensure strong real-time availability guarantees however, the Sancus platform should be extended with a secure hardware interrupt engine and generate non-maskable interrupts at fixed intervals. On such an interrupt, the hardware should store the program counter and CPU registers on the current private call stack, clear the CPU registers, save the stack pointer and vector to the SM_{sched} module. Sancus currently performs these operations by a compiler-inserted `sm_exit` stub when calling an external function, as explained in Sect. 3.1.7. In this respect, the `yield` calls in the current prototype simulate such an interrupt that transfers control to SM_{sched}. Implementing the `sm_exit` procedure in hardware is certainly possible and has successfully been done by Trustlite [30] and Intel SGX [36]. Moreover, De

[1] The technical details presented in this paragraph result from personal communication with Job Noorman.

Clercq et al. [14] describe a secure hardware interrupt engine that allows interrupting a task in a secure domain and vectoring to a non-secure domain, or the other way around.

An interesting future work direction would be to combine the Sancus threading model presented above with a modified hardware interrupt engine to construct secure logical control flows that will always be allowed to run, independent from any other logical threads being scheduled simultaneously. Masti et al. [35] describe such an embedded trusted scheduling architecture that ensures availability, even in the presence of malicious applications. From a security point of view however, their approach differs significantly from the one presented above in that they employ an omnipotent "trusted domain" software layer that is responsible for initialising the system and saving/restoring logical task state on context switch. Introducing such a kernel software layer in Sancus would of course invalidate the concept of hardware-protected SMs. Moreover, the SM_{sched} prototype demonstrates that an omnipotent kernel is not necessary to realise multithreading in a PMA.

Masti et al. [35] use an application-aware memory protection unit to enforce isolation of different tasks. Their architecture raises an exception on memory violation detection to transfer control to the protected domain. Furthermore, to be able to realise strong availability guarantees, a hardware mechanism transfers execution to the trusted domain at fixed time intervals. Analogously, the Sancus hardware should branch to SM_{sched} on a memory violation and at fixed time intervals through non-maskable interrupts, as explained in the previous section.

Masti et al. [35] also allow a programmer to mark code sections as atomic. To prevent an attacker from monopolising the CPU this way, they extend the hardware with an atomicity monitor to enforce that atomic code sections do not exceed a pre-set maximum length. An application that tries to execute an atomic code section exceeding this maximum value will be killed. A preemptive Sancus scheduler will certainly need to support such atomicity of code sections. An SM should for example not be preempted before it has restored its private call stack and authenticated its caller to ensure control flow integrity and access control restrictions such as those for the protected file system of Chapter 4. Analogously, an SM should be verified and called in a single atomic transaction, to avoid time-of-check-to-time-of-use vulnerabilities [47, 48] where a module is unloaded after it is verified, but before it is called.

Finally, Masti et al. [35] extend their availability guarantees with access control guarantees for an embedded peripheral bus. This way, they can ensure the on-schedule execution of critical applications, even in the presence of malicious applications and/or peripherals. Section 4.4.2 discussed how the generic access control mechanism for the file system could be employed to control access to a peripheral bus. The combination of the access control mechanism from the previous chapter with the threading model from this chapter could therefore provide SMs on a shared computing platform with strong real-time guarantees, enforced through a minimal protected TCB.

5.4.3 Comparison with Other PMAs

While the scheduler presented in this chapter was developed with the Sancus [40] platform in mind, the secure threading model is quite generic. Since multithreading is a known difficulty for PMAs, this section briefly contextualises the approach presented in this chapter.

The multitasking model of the Trustlite [30] architecture was already discussed in Sect. 5.1.2. The main difference with the model presented in this chapter is that Trustlite maps the unit of execution on the unit of protection. That is, Trustlite does not allow a thread of execution to span multiple protection domains. Instead, they consider protected modules as separate schedulable tasks and describe inter-module communication as Inter-Process Communication (IPC), which is usually a term that denotes cross-address-space communication with the help of a trusted OS. Trustlite features a single-address-space however, so calling a module comes down to jumping to the corresponding address, as in Sancus. One could therefore argue that logical threads that jump from module to module are the better option to represent control flow in a single-address-space. As opposed to the threading-aware protection domains introduced in this chapter, Trustlite's rather inconvenient IPC scheme for example requires a module to keep its own message buffers to queue subsequent calls. The treading model and scheduler implementation presented in this chapter could be ported to the Trustlite platform, on the condition that reliable and efficient caller authentication is provided. I.e. Trustlite would have to be extended with a `sancus_get_caller_id` alike hardware instruction.

Section 5.1.3 described the threading model of the SGX [36] architecture. SGX's threading model resembles that of this chapter in that protection domains are threading-aware and can choose for themselves how many internal logical threads they want to support. SGX is however intended for desktop and server computing and logical threads are scheduled by the untrusted OS as usual.

The Fides [48] hypervisor architecture, described in Sect. 2.2.3, realises protected modules in the virtual address space of a process through a separate secure virtual machine. As opposed to Sancus, SMs in Fides are an additional means of protection, isolated from the legacy OS. The legacy OS kernel remains responsible to schedule legacy threads that might enter an SM. When entering a module, the hypervisor pauses the legacy virtual machine and continues the execution of the current logical thread in the secure virtual machine. The threading model presented in this chapter is targeted at a single-address-space where SMs are the only means of protection, and thus makes little sense for Fides. The control flow integrity checks of Sect. 5.2.2 would make sense, since Fides modules can call and return to each other. To the best of the author's knowledge however, Fides does not feature reliable and efficient caller authentication, which is required to make sure that a callee only returns to its corresponding caller.

Fides [48] is targeted at high-end multi-core processors, but does not allow the concurrent execution of modules to avoid time-of-check-to-time-of-use vulnerabilities. Since Trustlite [30] allows modules to be interrupted at all times, time-of-check-to-time-of-use vulnerabilities could arise when interrupting a module that has verified,

but not yet called another module. Trustlite avoids these concerns by not allowing protected modules to be unloaded at run time.

5.5 Conclusion

This chapter presented a multithreading model and an accompanying protected scheduler implementation for the Sancus [40] platform. The prototype allows multiple logical control flow threads to co-exist simultaneously. Each such logical thread may run through multiple SM protection domains during its lifetime. To realise this in a secure way and without introducing an omnipotent kernel software layer that governs the system, participating SMs are made threading-aware. They are solely responsible for their internal control flow consistency, as enforced by guarding the entry of their protection domain. The prototype hides these concerns completely from the C programmer by small compiler-generated assembly code stubs.

The entry guards introduced in this chapter have furthermore resolved a number of vulnerabilities in Sancus' existing implicit control flow model and heavily rely on Sancus' caller authentication features.

Due to the lack of a secure hardware exception engine, the prototype does not feature a preemptive scheduler. The presented multithreading model and SM_{sched} module do however demonstrate the feasibility of such a preemptive scheduling environment that can enforce strong real-time availability guarantees. The prototype indeed allows the construction of secure logical threads that run through multiple fine-grained protection domains and that are conceptually isolated from each other.

Chapter 6

Conclusion

The introduction stated that small embedded devices are increasingly permeating our daily lives, whereas they commonly lack conventional security mechanisms. The success of these devices will therefore largely depend on the adequacy of novel lightweight software isolation techniques. In this perspective, embedded hardware-level PMAs [40, 30, 49] are a promising research direction as they allow efficient isolation of protected modules in a single-address-space. Without additional support however, hardware-level PMAs seclude software modules in their respective protection domains. That is, a protected module should either fulfil its own needs or rely on the services of an untrusted OS to interact with the outside world. This master's thesis has shown that the hardware-enforced security properties for protected modules can be extended with software-based availability and access control guarantees for shared platform resources.

This chapter is organised as follows. Section 6.1 acknowledges the prototype limitations and discusses the challenges that were encountered. Section 6.2 thereafter outlines future work directions and Sect. 6.3 concludes by summarising the contributions of this master's thesis.

6.1 Limitations and Challenges

This section acknowledges several prototype limitations and implementation obstacles.

First off, the work presented in this master's thesis heavily relies on Sancus [40] as the case study embedded PMA. While the underlying ideas of the access control mechanism and scheduling model are quite generic, they inevitably rely on sufficient hardware support. Sections 4.4.4 and 5.4.3 briefly discussed how the respective approaches could be ported to other PMAs.

To arrive at the current protected file system, some implementation challenges were encountered. Section 4.2.1 briefly discussed an early file system prototype that was abandoned for several reasons. First, its internal UNIX-like inode-based structure neglected flash-specific properties and was therefore barely usable for the peripheral flash drive. Moreover, by storing access control meta data in the on-disk inode table, the prototype anticipated persistent file protection, which turned out

to be non-trivial. Finally, the move towards a layered architecture that separates the access control logic from the actual file system implementation, allowed for the lightweight front-end to be re-used for the protected shared memory implementation.

An important limitation of the current file system prototype is its lack of persistent file protection. Several strategies to achieve persistence were discussed in Sect. 4.4.3, but each of them has its own downsides and further research is needed to investigate their performance and security implications.

The creation of meaningful macro benchmarks for the file system set up was prevented because the Coffee [51] file system operates in the unprotected domain, as acknowledged in Sect. 4.4.3. Protecting the Coffee file system was attempted, but abandoned since Coffee's micro logging code is buggy and uses features that are not directly supported by the Sancus compiler.

The multithreading prototype also has several drawbacks. To start with, the complexity of the compiler-generated assembly entry stubs is increased. Especially since they have to link against the scheduler. While these stubs are hidden from the C programmer, he still needs to provide the address, SMID and EIDXs of the scheduler. Moreover, the `get_cur_thr_id` call on every entry decreases performance. The current prototype furthermore neglects the unprotected domain, by not allowing it to yield to the scheduler. Further research is needed to investigate the availability, control flow and security implications of suspending and continuing the unprotected domain.

6.2 Future Work

The secure access control mechanism as well as the secure threading model can be improved in several ways.

Persistent File Protection A first future work possibility is to support persistent file protection. Several strategies were discussed in Sect. 4.4.3, but further research is needed to investigate their security and performance implications. The application scenarios and desired security guarantees must thereby be taken into account. To securely store a module's state on untrusted storage, existing work on state continuity [42] could be considered.

Hardware Support for Shared memory Section 4.3 revealed a considerable overhead for the protected shared memory implementation. It was furthermore showed that safely transferring data between SMs on a byte-per-byte basis through CPU registers, is responsible for the majority of the access control overhead. Native hardware support for a simple form of protected shared memory between a caller and a callee, as proposed in Sect. 4.4.4, could therefore drastically improve performance of the protected file system.

SPI Bus Access Control Section 4.4.2 briefly discussed how the generic resource sharing approach could be employed to control access to a SPI peripheral bus. Limiting access to peripherals would be valuable to operate SMs under the

principle of least privilege [44]. That is, a potentially malicious module is only allowed access to the peripherals it is assigned by the access control policy. This allows availability guarantees in the presence of malicious modules, analogous to the approach of Masti et al. [35].

Multithreaded Protection Domains Section 5.3.3 explained how SMs were made threading-aware and how they could feature multiple internal logical threads. In the current prototype however, SMs execute for at most one thread at all times. When adding support for multiple internal logical threads, the corresponding call stacks should be properly separated. Moreover, support for mutexes should be implemented to allow synchronisation of internal logical threads. Consider for example a multithreaded protected file system where consistency of the shared access control data structures is essential.

Hardware Exception Engine Section 5.4.2 discussed how extending Sancus' hardware logic with a secure hardware exception engine allows for a preemptive scheduler. Such a scheduler can provide safety-critical embedded systems with strong real-time availability guarantees, even in the presence of malicious modules.

Atomicity Constraints Further research is needed to investigate the effect of preemption on the correct execution of protected modules. It will thereby be essential to provide adequate support for atomicity of code sections, while at the same time preventing an attacker to monopolise the CPU time. A software module should for example not be preempted before it has restored its private call stack and authenticated its caller to perform the necessary access control checks, introduced in Sect. 5.2.2. Other well-known [47, 48] multithreading issues for PMAs include time-of-check-to-time-of-use vulnerabilities, where thread A authenticates a module and is preempted before making the call. The already authenticated module can now be unloaded in another thread B, so that A jumps to unprotected code.

Suspending the Unprotected Domain A final consideration involves the security implications of interrupting a thread that is executing in the unprotected domain. While the unprotected domain could easily be made internally multithreaded, there is no way to enforce this separation. That is, anyone can access the unprotected call stack of an interrupted thread, breaking logical isolation of different threads. An attacker executing in thread B can indeed influence the control flow of another thread A that was previously interrupted in the unprotected domain. Further research is therefore needed to for example isolate widely used `libc` functions in their own protection domain.

6.3 Contributions

This master's thesis explored the possibility of securely implementing OS-like resource sharing abilities on top of embedded hardware-level PMAs. Using Sancus [40] as the

development platform, the following contributions were made:

- The minimal set of hardware primitives that need to be provided by a PMA to securely allow the software implementation of OS-like services was identified. These primitives are (*i*) memory isolation, (*ii*) module authentication, (*iii*) caller authentication, and (*iv*) exclusive use of MMIO ranges. Especially efficient caller authentication proved to be a prerequisite to implement access control, as well as to realise control flow guarantees. Chapter 3 compared Sancus' hardware primitives to those of a minimalist microkernel [33] and Chapter 5 identified the need to extend the hardware with a secure exception engine.

- Chapter 4 presented a protected file system implementation as a case study of encapsulating and controlling access to a typical shared system resource. The file system provides clients with the concept of a protected logical file and realises flexible SM-grained access control policies. Two back-ends were implemented, allowing access control to either a protected shared memory buffer or a peripheral flash drive with the Coffee [51] file system. The security guarantees and general applicability of the resource sharing mechanism were discussed.

- Sancus' existing implicit control flow model was secured by inserting extra compiler-generated runtime checks at the boundaries of protected modules, as described in Sect. 5.2.2.

- Chapter 5 furthermore presented a multithreading model and protected scheduler implementation to control access to the CPU time resource. The scheduler decouples the unit of execution from the unit of protection by interweaving the execution of multiple logical threads that might span arbitrarily many SM protection domains. Moreover, SMs were made threading-aware so that they only rely on the scheduler for availability guarantees and remain solely responsible for their internal control flow and data. All these concerns are completely hidden from the C programmer by compiler-generated assembly code stubs.

The source code of the protected file system and the secure scheduler is publicly available at `https://github.com/jovanbulck/thesis-src`.

Appendices

Appendix A

Complete Threading Example

This appendix presents a minimal, yet complete example of a multithreaded program. As such, it demonstrates the protected scheduler from Chapter 5. Listing A.1 lists the C source code of the example program. It defines two modules SM_{foo} and SM_{bar}, each serving as the portal of a logical thread. The `main` function first enables these modules. Thereafter, it calls the `set_foo_vars` and `set_bar_vars` functions. These functions initialise the private variables from Sects. 5.2.2 and 5.3.3 to be used from the compiler-generated `sm_entry` assembly code stubs. To ensure secure linking against the scheduler, these functions go into an infinite loop if SM_{sched} was not successfully verified.

The corresponding sequence diagram is shown in Fig. A.1 and illustrates threading-aware protection domains. It shows how the scheduler first starts the light gray logical thread through the SM_{foo} portal. When the current thread yields, the scheduler starts the dark gray thread through the SM_{bar} portal, which subsequently tries to call SM_{foo}. This module is currently already executing for the light gray thread and remarks through the scheduler's `get_cur_thr_id` function that the currently executing thread is the dark gray one. As explained in Sect. 5.3.3, participating SMs have to ensure internal monothreading. SM_{foo} therefore returns the magic busy value in the agreed CPU register. The calling SM_{bar}'s assembly entry stub notifies this and automatically yields to the scheduler, which continues the light gray thread. When this thread finally returns, the dark gray one is continued. SM_{bar}'s assembly entry stub automatically retries the previous call, which succeeds this time since SM_{foo} is not working for the light gray thread any more. Finally, SM_{bar} returns to the scheduler, which notifies all logical threads have completed their execution and returns to the `main` function.

As demonstrated by Listing A.1, the control integrity and internal monothreading guards discussed in Sects. 5.2.2 and 5.3.3 are completely hidden from the programmer. They are indeed entirely implemented by the small assembly code stubs inserted at compile time. The C programmer simply codes the logical control flow, using as many SM protection domains as he likes, and finally identifies the logical threads through their portal SMs.

Listing A.1: C source code of a simple multithreaded program

```
1  #include <sancus/sm_support.h>
2  #include "scheduler.h"
3
4  DECLARE_SM(foo, 0x1234);
5
6  void SM_ENTRY("foo") set_foo_vars(void)
7  {
8      __sm_foo_sad = scheduler.public_start;
9      __sm_foo_sid = sancus_verify(SM_GET_TAG(foo, scheduler), &
           scheduler);
10     __sm_foo_vep = (entry_idx) &
           __sm_scheduler_entry_report_entry_violation_idx;
11     __sm_foo_yep = (entry_idx) &__sm_scheduler_entry_yield_idx;
12     __sm_foo_gep = (entry_idx) &
           __sm_scheduler_entry_get_cur_thr_id_idx;
13
14     if(!__sm_foo_sid)
15         while(1) {}
16 }
17
18 void SM_ENTRY("foo") start_foo(void)
19 {
20     // do things
21     yield();
22     // do more things
23     return;
24 }
25
26 void SM_ENTRY("foo") call_foo(void)
27 {
28     // do things
29     return;
30 }
31
32 DECLARE_SM(bar, 0x1234);
33
34 void SM_ENTRY("bar") start_bar(void)
35 {
36     // do things
37     call_foo();
38     // do more things
39     return;
40 }
41
42 void SM_ENTRY("bar") set_bar_vars(void)
43 {
44     __sm_bar_sad = scheduler.public_start;
45     __sm_bar_sid = sancus_verify(SM_GET_TAG(bar, scheduler), &
           scheduler);
46     __sm_bar_vep = (entry_idx) &
           __sm_scheduler_entry_report_entry_violation_idx;
47     __sm_bar_yep = (entry_idx) &__sm_scheduler_entry_yield_idx;
48     __sm_bar_gep = (entry_idx) &
```

```
        __sm_scheduler_entry_get_cur_thr_id_idx;

    if(!__sm_bar_sid)
        while(1) {}
}

int main()
{
    sancus_enable(&foo);
    sancus_enable(&bar);
    sancus_enable(&scheduler);

    set_foo_vars();
    set_bar_vars();

    register_thread_portal(&foo, SM_GET_ENTRY_IDX(foo, start_foo));
    register_thread_portal(&bar, SM_GET_ENTRY_IDX(bar, start_bar));

    start_scheduling();

    while(1) {}
}
```

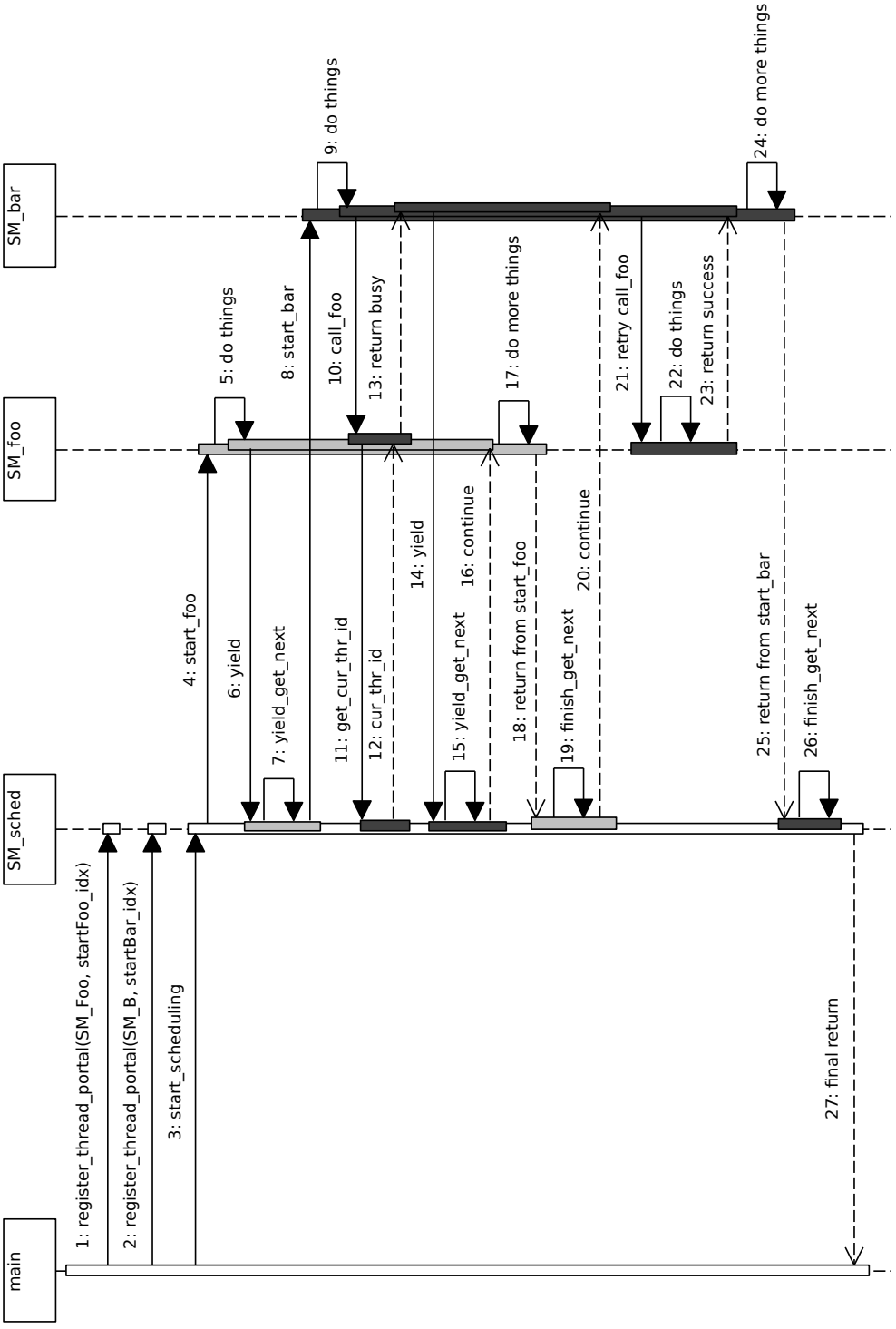

Figure A.1: Sequence diagram of the program from Listing A.1 to demonstrate threading-aware protection domains. For readability purposes, the figure only visualises the relevant `get_cur_thr_id` call, instead of on every SM entry.

Bibliography

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353. ACM, 2005.

[2] P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. In *Computer Security Foundations Symposium*, pages 171–185. IEEE, 2012.

[3] N. Avonds. Implementation of a state-of-the-art security architecture in the linux kernel. Master's thesis, KU Leuven, 2013.

[4] M. J. Bach. *The design of the UNIX operating system*, volume 5. Prentice-Hall, 1986.

[5] A. Berman, V. Bourassa, and E. Selberg. Tron: Process-specific file protection for the unix operating system. In *USENIX*, pages 165–175, 1995.

[6] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He. The liteos operating system: Towards unix-like abstractions for wireless sensor networks. In *International Conference on Information Processing in Sensor Networks*, pages 233–244. IEEE, 2008.

[7] N. P. Carter, S. W. Keckler, and W. J. Dally. Hardware support for fast capability-based addressing. In *ACM SIGPLAN Notices*, volume 29, pages 319–327. ACM, 1994.

[8] J. S. Chase, H. M. Levy, M. Baker-Harvey, and E. D. Lazowska. Opal: a single address space system for 64-bit architecture address space. In *Workstation Operating Systems, 1992. Proceedings., Third Workshop on*, pages 80–85. IEEE, 1992.

[9] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, 1994.

[10] T. V. Chien, H. N. Chan, and T. N. Huu. A comparative study on operating system for wireless sensor networks. In *Advanced Computer Science and Information System (ICACSIS), 2011 International Conference on*, pages 73–78. IEEE, 2011.

[11] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for TinyOS. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 205–218. ACM, 2007.

[12] J. Corbet, G. Kroah-Hartman, and A. McPherson. Linux kernel development. how fast it is going, who is doing it, what they are doing, and who is sponsoring the work. `https://www.linuxfoundation.org/publications/linux-foundation/who-writes-linux-2015`, February 2015.

[13] A. Cui and S. J. Stolfo. A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 97–106. ACM, 2010.

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

[15] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 15–28. ACM, 2006.

[16] A. Dunkels, N. Tsiftes, Z. D. Purvis, and R. Léone. Contiki OS wiki: File systems. `https://github.com/contiki-os/contiki/wiki/File-systems`, last updated December 10 2014.

[17] A. Dwivedi, M. Tiwari, and O. Vyas. Operating systems for tiny networked sensors: A survey. *International Journal of Recent Trends in Engineering*, 1(2):152–157, 2009.

[18] Ú. Erlingsson, Y. Younan, and F. Piessens. Low-level software security by example. In *Handbook of Information and Communication Security*, pages 633–658. Springer, 2010.

[19] S. Escolar, J. Carretero, F. Isaila, and S. Lama. A lightweight storage system for sensor nodes. In *PDPTA*, pages 638–644, 2008.

[20] M. O. Farooq and T. Kunz. Operating systems for wireless sensor networks: A survey. *Sensors*, 11(6):5900–5930, 2011.

[21] A. A. Fröhlich and L. F. Wanner. Operating system support for wireless sensor networks. *Journal of Computer Science*, 4(4):272–281, 2008.

[22] D. Gay. Matchbox: A simple filing system for motes. `http://www.docs.tinyos.net/tinyos-1.x/doc/matchbox.pdf`, August 21 2003.

[23] A. Grünbacher. Posix access control lists on linux. In *USENIX Annual Technical Conference, FREENIX Track*, pages 259–272, 2003.

[24] L. Gu and J. A. Stankovic. t-kernel: Providing reliable os support to wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 1–14. ACM, 2006.

[25] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, May 2009.

[26] IEEE and The Open Group. chmod ieee std 1003.1. `http://pubs.opengroup.org/onlinepubs/009695399/functions/chmod.html`, 2004.

[27] Intel Coporation. Intel software guard extensions programming reference. `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`, October 2014.

[28] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[29] K. Klues, C.-J. M. Liang, J. Paek, R. Musaloiu-Elefteri, P. Levis, A. Terzis, and R. Govindan. Tosthreads: thread-safe and non-invasive preemption in tinyos. In *SenSys*, volume 9, pages 127–140, 2009.

[30] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. Trustlite: a security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 10:1–10:14. ACM, 2014.

[31] P. Koopman. Embedded system security. *Computer*, 37(7):95–97, 2004.

[32] R. Kumar, A. Singhania, A. Castner, E. Kohler, and M. Srivastava. A system for coarse grained memory protection in tiny embedded processors. In *Design Automation Conference*, pages 218–223. IEEE, 2007.

[33] J. Liedtke. On μ-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 237–250. ACM, 1995.

[34] J. Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, 1996.

[35] R. J. Masti, C. Marforio, A. Ranganathan, A. Francillon, and S. Capkun. Enabling trusted scheduling in embedded systems. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 61–70. ACM, 2012.

[36] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 10:1–10:1. ACM, 2013.

[37] Microsoft. A history of windows. `http://windows.microsoft.com/en-AU/windows/history#T1=era6`, November 2013.

[38] J. T. Mühlberg and G. Lüttgen. Blasting Linux Code. In *11th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, volume 4346, pages 211 – 226. Springer, 2006.

[39] J. Noorman. Sancus 1.0 source code. `https://distrinet.cs.kuleuven.be/software/sancus/`, 2013.

[40] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX SEC'13*, pages 479–494. USENIX Association, 2013.

[41] S. Parameswaran and T. Wolf. Embedded systems security – an overview. *Design Automation for Embedded Systems*, 12(3):173–183, 2008.

[42] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 379–394. IEEE, 2011.

[43] W. Penninckx, J. T. Mühlberg, J. Smans, B. Jacobs, and F. Piessens. Sound formal verification of Linux's USB BP keyboard driver. In *NASA Formal Methods*, pages 210–215. Springer, 2012.

[44] J. H. Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402, 1974.

[45] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[46] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley, 8 edition, 2010.

[47] R. Strackx, J. Noorman, I. Verbauwhede, B. Preneel, and F. Piessens. Protected software module architectures. In *ISSE 2013 Securing Electronic Business Processes*, pages 241–251. Springer, 2013.

[48] R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 2–13. ACM, 2012.

[49] R. Strackx, F. Piessens, and B. Preneel. Efficient isolation of trusted subsystems in embedded systems. In *Security and Privacy in Communication Networks*, pages 344–361. Springer, 2010.

[50] A. S. Tanenbaum and A. S. Woodhull. *Operating systems: design and implementation*, volume 3. Pearson, 2009.

[51] N. Tsiftes, A. Dunkels, Z. He, and T. Voigt. Enabling large-scale storage in sensor networks with the coffee file system. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, pages 349–360. IEEE Computer Society, 2009.

[52] S. Underwood. mspgcc: A port of the gnu tools to the texas instruments msp430 microcontrollers. `http://mspgcc.sourceforge.net/manual/x1248.html`, 2003.

[53] J. Viega and H. Thompson. The state of embedded-device security (spoiler alert: It's bad). *IEEE Security & Privacy*, 10(5):68–70, 2012.

[54] E. Witchel, J. Rhee, and K. 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. ACM, 2005.

[55] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The cheri capability model: Revisiting RISC in an age of risk. In *Proceeding of the 41st annual international symposium on Computer architecuture*, pages 457–468. IEEE Press, 2014.

[56] Y. Yao, L. Wan, and Q. Cao. System architecture and operating systems. In *The Art of Wireless Sensor Networks*, pages 697–738. Springer, 2014.

EOF

Fiche masterproef

Student: Jo Van Bulck

Titel: Secure Resource Sharing for Embedded Protected Module Architectures

Nederlandse titel: Veilig Delen van Systeembronnen voor Geïntegreerde Beveiligingsarchitecturen

UDC: 681.3

Korte inhoud:

Small embedded devices are becoming omnipresent in our daily lives. Yet, to minimise production costs and power consumption, these devices commonly lack hardware support for conventional security measures, such as virtual memory and processor privilege levels.

In this respect, recent research on hardware-level Protected Module Architectures (PMAs) provides an alternative, very lightweight memory protection scheme. These systems allow the execution of security-critical code in protected modules that are isolated from the rest of the system, without relying on a trusted software layer to enforce this separation. While secluding software modules in their own hardware-enforced protection domains allows for strong security guarantees, it also limits their ability to securely share platform resources, such as CPU time or peripheral devices.

This master's thesis explores the feasibility of supplementing the hardware-enforced security guarantees offered by the Sancus PMA with availability and access control guarantees for shared system resources. In contrast to a conventional Operating System (OS), an omnipotent kernel software layer is not introduced. The main contributions of this master's thesis are twofold. First, a generic approach to encapsulate and control access to a shared platform resource is proposed. The approach is implemented and evaluated for a protected file system that can control access to either a shared memory buffer or a shared peripheral flash drive. Second, a secure multithreading model and an accompanying unprivileged scheduler implementation are presented. The scheduler controls access to the CPU time resource by interweaving the execution of logical threads that are conceptually isolated from each other and that might span multiple protection domains.

The work presented in this master's thesis shows that embedded PMAs provide sufficiently strong hardware primitives to not only isolate software modules from each other, but also allow secure implementation of typical OS responsibilities.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdspecialisatie Veilige software

Promotor: Prof. dr. ir. Frank Piessens

Assessoren: Dr. ir. C. Huygens
Dr. J. Sneyers

Begeleiders: Ir. J. Noorman
Dr. J.T. Mühlberg