Alexander Köberl, BSc

Implementation and Design of an Advanced Memory Manager based on the ARMv8-M Security Extension

MASTER THESIS

to achieve the university degree of

Master of Science

Master's degree programme: Information and Computer Engineering

submitted to:

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger

Institute for Technical Informatics

Advisor

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger

Dipl.-Ing. Dr.techn. Ulrich Neffe (NXP Semiconductors Austria GmbH)

Graz, December 2018

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

Kurzfassung

Mit der steigenden Leistung von Mikrokontrollern ergeben sich neue Anwendungsfälle und die Komplexität der eingesetzten Software vergrößert sich. Der Aufschwung in der Internet of Things (IoT) Branche führt zu einer starken Verbreitung von kleinen und verbundenen Geräten in unserem Alltag. Durch das Hinzufügen von Konnektivität zu diesen einfachen Geräten, sind die Auswirkungen der Software nicht mehr auf die unmittelbare Umgebung begrenzt.

Der erweiterte Funktionsumfang und die gesteigerte Leistung machen sie zu einer guten Wahl für energieeffiziente und preisgünstige Lösungen. Projekte, die früher einen komplexen Prozessor benötigten, können jetzt auf einem einfachen Mikrokontroller realisiert werden.

Diese neuen Anwendungsfälle stellen neue Anforderungen an die Systemarchitektur und benötigen einen stärkeren Fokus auf Sicherheit und Vertraulichkeit. Leider ist die zugrundeliegende Hardwarearchitektur von Mikrokontrollern nicht speziell auf sichere Software ausgelegt. Kurze Interrupt Latenzen und ein deterministischer Programmablauf haben höhere Priorität in eingebetteten Systemen. Die *Cortex-M* Familie im speziellen, benutzt noch immer ein flaches Speichermodell ohne zusätzliche Unterstützung einer Memory Management Unit (MMU). Der aktive Programmteil kann auf alle Daten und Hardwarekomponenten zugreifen, auch wenn eine Applikation in mehrere Tasks unterteilt wird.

Ein erfolgreicher Angriff auf eine Systemkomponente gefährdet auch unbeteiligte Teile. Zusätzlich kann ein Programmierfehler auch Seiteneffekte in streng verifizierten Tasks auslösen. Um die Korrektheit von wenigen kritischen Komponenten zu garantieren, muss eine aufwändige Validierung des kompletten Systems durchgeführt werden.

Die *Security Extension* der neuen *ARMv8-M* Architektur kann dazu beitragen, die Systemsicherheit zu verbessern. Eine strenge Unterteilung in sichere und unsichere Speicherbereiche wird von Hardwarekomponenten durchgesetzt. Eine Partitionierung ist jedoch nur in zwei Sicherheitsdomänen möglich: Alle abgesicherten Komponenten können nicht zusätzlich voneinander abgegrenzt werden.

Diese Arbeit konzentriert sich auf die Erweiterung des Systems, um eine zusätzliche Unterteilung der Zugriffsberechtigungen in der sicheren Domäne durchzusetzen. Ein Kontextmanager verwaltet Speicherbereiche und aktiviert benötigten Code und Daten dynamisch. Diese Isolierung verhindert einen uneingeschränkten Systemzugriff durch sicheren Code. Eine Kombination aus den Funktionen der *Security Extension* und einer proprietären Bus-Firewall von *NXP* wird für die Realisierung benutzt.

Das vorgeschlagene Design bietet flexible Rahmenbedingungen für die eingefügten Programmteile. Bereits vorhandene Projekte und Bibliotheken von externen Quellen benötigen nur wenige Anpassungen. Alle Übergänge zwischen den Sicherheitsdomänen werden unterstützt um die Funktionalität des Anwendungscodes nicht einzuschränken.

Abstract

With the increasing performance of embedded microcontrollers, new use cases arise and the complexity of the deployed software grows. The recent boom of the Internet of Things (IoT) market lead to a spreading of small and connected devices in our everyday life. By adding connectivity to those simple systems, the software is no longer confined by the device borders.

The added hardware features and performance increase make them a feasible choice for energy efficient and low-cost designs. Projects, which previously required an application processor, can now be deployed on a microcontroller. This often entails performing of safety critical operations and handling of sensitive data.

Those new applications have a stronger focus on security and confidentiality, which requires a change in the system architecture. Unfortunately, the underlying hardware architecture of microcontrollers was not designed with software security as main priority. Low interrupt latency and deterministic execution of the firmware is a stronger requirement of embedded systems. Focusing on the *Cortex-M* family, there is still a flat memory model without an additional Memory Management Unit (MMU) in place. When an application is split into distinct tasks, the active code can access unrelated data and hardware components without restrictions.

Compromising a single component can expose the complete system to an adversary. Additionally, a programming fault can lead to corruption of otherwise strictly validated tasks. This requires extensive verification of the complete system to guarantee the correctness of only a small part of critical operations.

The *Security Extension* of the new *ARMv8-M* architecture is an improvement for system security. Strict separation of secure and non-secure memory areas is enforced by the hardware architecture of the core. Nevertheless, it only provides a partitioning into two security domains: All secure code is part of the same Trusted Code Base (TCB) with the previously mentioned security concerns.

This thesis focuses on extending the present system, to further limit the access permissions within the secure domain. A context manager keeps track of distinct memory sections and dynamically activates required code and data. With this isolation, secure code has no longer full permission over the complete system. This protection is realized by combining the *Security Extension* with a proprietary bus-firewall developed by *NXP*.

The proposed design provides an unobtrusive framework for deployed code. Required modifications of legacy code and third-party libraries are kept to a minimum. All transition sequences between the security domains are supported, to impose no restrictions on the application code.

Acknowledgements

This thesis was developed in cooperation with *NXP Semiconductors Austria GmbH.* I would like to thank Dipl.-Ing. Dr.techn. Ulrich Neffe for initiating and advising on the project, and Dipl.-Ing. (FH) Christian Eisendle for the great support and guidance. Furthermore, thanks to Dipl.-Ing. Lukas Gressl for the suggestions on writing style and the in-depth review. It was a unique opportunity, to work on such a recent research-topic with the support of a leading company.

Finally, thanks to Prof. Christian Steger and the Institute for Technical Informatics for the strong cooperation with *NXP* and approving this thesis. With the past experience in the research field, valuable recommendations were given during development and refinement of the resulting documentation.

Graz, December 2018 Alexander Köberl

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter gives the motivation for the project by introducing the current requirements of the embedded-system market. The focus is set at device security and the protection of Intellectual Property (IP).

The concrete project goals are defined to give an overview of the proposed architecture and list the needed features. An example software partitioning for a payment terminal is given to introduce the target use case.

1.1 Motivation

Embedded systems were originally deployed in self-contained devices to complete a certain technical process, controlled by a local user interface. With the increasing number of connected micro controllers caused by the steady growth of the Internet of Things (IoT) market, our surrounding is flooded with remote-controlled sensors and actuators. New use cases call for cheap, reliable, secure and power-efficient devices, creating challenges for engineers and manufacturers.

Another trend is the increasing complexity and specialization of System-on-Chips (SoCs). Hardware modules for specific applications are integrated directly on the silicon of the microcontroller, to reduce the overall footprint and increase performance. Examples include connectivity, cryptography modules, coprocessors and interfaces for cameras and displays. Customers expect high-level Application Programming Interfaces (APIs) and hardware abstraction for shorter development cycles and increased portability. The chip manufacturer might not want to disclose the internal implementation of those libraries to protect IP or prevent tampering of certified components.

The following sections highlight the need for an advanced security architecture for embedded devices regarding security and protection of IP.

1.1.1 Device security

Mass produced IoT devices recently became a popular target for attacks. This could be motivated by the high availability of cheap devices, a potentially high profit of a successful attack and the limited security features available on those low-power devices.

When a device is attached to a network, it can be a victim of a targeted attack, or become a gateway to cause harm to other connected members. In the former case, an adversary uses a remotely accessible vulnerability of the device to manipulate the functionality or gain access to confidential data.

One of the most famous examples is the *Stuxnet* worm which targeted Programmable Logic Controllers (PLCs) used in industrial machines [13]. In one case, it was deployed to modify the speed of centrifuges used for the Iranian nuclear program, which caused premature wearing and damage to the machines. Other established targets are internet-enabled surveillance cameras and routers to attack the privacy of a target.

A new application for compromised embedded devices emerged in 2016, when the *Mirai* malware infected Digital Video Recorders (DVRs), smart TVs and routers. The adversary later used those devices to overload large websites with Distributed Denial of Service (DDoS) attacks [12]. Another theoretical attack scenario is the usage of high-wattage IoT devices (e.g. air conditioning, heaters) to destabilize the power grid to the point of a local blackout [26].

Although the previously mentioned attacks were mostly enabled by misconfiguration or bugs on higher level software, the impact might have been reduced by using a secure execution environment for critical system components. Another line of defence can be implemented by adding an integrity monitor, which continuously verifies the system and resets to a known state when unauthorized modifications are detected [27]. A trusted execution environment is also required when adding a simple interface for secure firmware upgrades.

Security vulnerabilities are often uncovered by reverse engineering the software image and tracing the external signals. To close this attack vector, encrypted binaries (with hardware secured keys) and reliable encryption of external communication is required.

Both of these aspects can be implemented as trusted services of a secure execution environment. Additionally, debugging of secure code can be prevented for release versions of the product. This prohibits inspection of the software binary and access to the internal processor state via register values during execution.

1.1.2 Intellectual property protection

Complex devices are mostly developed by a chain of individual companies. Figure 1.1 illustrates the sequence of manufacturing steps in this case.

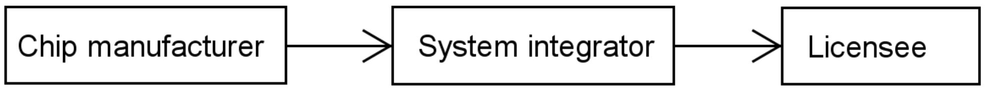

Figure 1.1: Cooperative development steps

Every participator in the manufacturing chain has different requirements regarding confidentiality and available system features:

- **Chip manufacturer:** Delivers the SoC with complex hardware drivers. This IP might contain a lot of know-how and must be protected from reverse engineering and unauthorized modifications.

- **System integrator:** Needs an easy-to-use API to the low-level drivers. Develops middleware to external components and the main application which might require extensive verification and certification. Secure storage is required to protect keys and other secrets from tampering by subsequent suppliers and the end customer.
- **Licensee:** Customizes an off-the-shelf device with minimal interference to the base system. Changes are mostly cosmetic, e.g. Graphical User Interface (GUI) branding, and should not require further certification efforts.

This example illustrates the necessity of software isolation to protect IP on multiple stages of development.

1.2 Goals

The goal of this project is to implement a secure component, for isolating critical memory sections from unauthorized accesses. Sandboxing concepts from other application domains are evaluated, to derive ideas for this proof of concept design. The new *Security Extension* of the *ARMv8-M*-architecture and the proprietary Advanced High-performance Bus (AHB) firewall, developed by *NXP*, are used to enforce access permissions.

Individual modules of the firmware are divided into distinct memory sections, which are called *contexts* in the remainder of this document. Different unrelated libraries or tasks of an operating system could be separated this way.

The purpose of those isolated contexts is to provide protection from intentional and accidental corruption of internal data on the one hand. On the other hand, multiple parties should be able to contribute functions to the secure domain without fear of disclosing their IP.

As described in Section 1.3, the target use case of the proposed design is a next generation of Point of Sale (POS)-terminals. The SoC for this application typically includes Near Field Communication (NFC) hardware, which is abstracted by a complex API, and security critical payment libraries.

A company confidential hardware prototype is used as target platform. It is developed for a different application but features all relevant security modules. Issues, which are discovered during this work, will be resolved during the design of the final product.

The proposed architecture must support the following key features:

- Provide protection for individual contexts (e.g. libraries or tasks) within the secure execution environment.
- Allow flexible cross-domain calls without the need of explicit supervisor or Remote Procedure Calls (RPCs).
- The non-secure domain should not be influenced by the added software components to enable the simple porting of Real Time Operating Systems (RTOSs) and legacy projects.
- The design should allow the addition of secure functions from a third-party without the need of recompiling the entire secure code.

1.3 Example Use Case

An example use case, which combines both challenges mentioned in Section 1.1, is the architecture of a payment terminal. Although context isolation should also be applied to other application domains (e.g. IoT), the rest of the document is focused on this implementation. Figure 1.2 illustrates an example firmware partitioning of such a device, which incorporates modules from different suppliers. A detailed description of a POS architecture is given in Section 2.4.

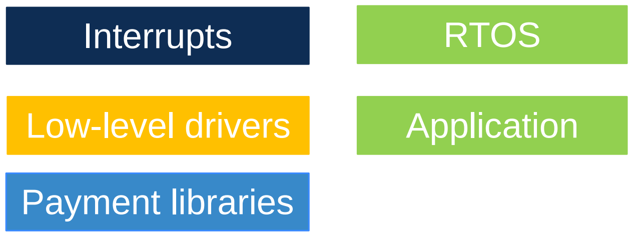

Figure 1.2: Software components of a payment terminal

- Low-level drivers from the chip manufacturer include access to the card interface and complex interaction with the Contactless Interface (CLIF).
- The system integrator adds payment libraries and uses the low-level drivers to interface external hardware components (e.g. key pads).
- A final licensee might develop or modify an existing payment application. Projects with this complexity generally use a RTOS and split the execution into distinct tasks.

1.4 Structure

Prerequisites to give background knowledge are given in Chapter 2. It starts with giving an introduction to the Memory Protection Unit (MPU) of the *Cortex-M* microcontroller series. This is required to understand the common memory protection technique, which is heavily used in projects for this architecture. The limitations of this module are pointed out to emphasize the need for a new approach to tackle advanced security scenarios.

Afterwards, the new *Security Extension* of the *ARMv8-M* architecture is described in more detail. It is one of the key features enabling the functionality of this project. The provided security-domain transitions with all side effects and limitations are examined, to get the background knowledge for the design and implementation of the context manager. An overview of the included secure debug functionality is given to conclude the added security functionality of the new architecture.

A large part of the available literature does not make a distinction between the *Cortex-A* and *Cortex-M* cores. The primary features of the former are introduced to make this classification easier and give the main differences. Another source for confusion is the synonym

'TrustZone' for the new security extension. It is used on both architectures for similar concepts, but the core functionality follows different approaches. Section 2.3.1 describes the distinct attributes of both systems.

To conclude the preliminaries, an overview of a POS architecture is given. It will be the target use case of the developed solution of this project. Attack vectors and currently employed security measures are mentioned to highlight the need for an improved security solution for payment terminal firmware. The security standard of this industry is also mentioned to further confirm the importance of the proposed context manager.

Related work is discussed in Chapter 3. Evaluation of RTOS projects is done in the first section. Security features and established concepts are discussed based on practical examples. The second part looks at applications of the *TrustZone* of *Cortex-A* processors. Finally, the *ARM Platform Security Architecture*, with focus on *Trusted Firmware-M*, is introduced as a potential solution for reducing vulnerabilities in IoT devices.

The relevant features, which could be reused for this project, are summarized. Those possible design choices are discussed, and apparent restrictions are outlined.

Chapter 4 is concerned with the design of the context manager. It starts with defining the main requirements of the proposed solution. The technical details of the target hardware platform are listed to present the influence of the specific architecture to the design. Based on this information, a memory map defining the secure and non-secure sections is defined. The prepared software partitioning is later used at the implementation stage.

The main part of this chapter is spent on the design of the context manager itself. The high-level operation flow is further refined by giving the detailed sequences of the different context transitions. Important aspects of defining the sanity check, for monitoring calls within the secure domain, are also given.

The implementation is described in more detail in Chapter 5. The used toolchain, required development steps and the project architecture are also documented there.

Detailed instruction sequences illustrate the different context transitions. They show steps which are performed implicitly by the processor core and give instructions about identifying them. The required tasks, which have to be performed by the context manager are also specified.

Finally, the method for validating the resulting implementation is described.

The evaluation in Chapter 6 starts with the timing behavior of the context switches. Detailed measurements were taken and discussed.

Another focus is set on the different attack vectors which are handled by the different components. It also gives best-practice examples and security recommendations.

Finally, the conclusion in Chapter 7 compares the results of the project to the initial requirements. Proposed design changes of the target platform are described and recommendations for improving the context manager are also given. Meaningful extensions, which were out of the project scope, are mentioned for future expansion.

Chapter 2

Preliminaries

The following sections give a brief introduction to the relevant features of the *ARM* architecture and general embedded security concepts, which are used for evaluation and implementation of the final solution. They provide a quick overview of the used terms which are needed to understand the remainder of this document, but are not meant to be an advanced reference. Additionally, the specification of the target hardware is presented.

2.1 Memory Protection Unit

Cortex-M processors provide two execution modes: Privileged execution for full access to all registers during start-up or in exception handlers, and unprivileged execution for application code.

The Memory Protection Unit (MPU) can be further used to define the allowed privilege level for memory accesses [5]. Before the introduction of *TrustZone* to embedded controllers, this was the only measure provided by the ARM design to prevent accesses to restricted memory areas. When an invalid read/write or execution fetch is performed, a fault exception is triggered.

Being an optional feature, the number of distinct configurable regions ranges from 0 up to 16. For *ARMv8-M*-series processors, the MPU is configurable independent for each security state, if it is implemented on the specific SoC.

Every section can be used to define the permission settings for a continuous address range from 32 bytes to 4 gigabytes. The value of the ***A****ccess* ***P****ermissions* field of the configuration register selects the permission as described in Table 2.1.

Additionally, sections can be marked as *Execute Never* to provide a measure against arbitrary code injection through user data. This restricts the freedom of an attacker when exploiting return-oriented programming [32].

This basic security feature is often utilized for task isolation in an RTOS. The kernel of the operating system and interrupt handler code is hidden from the unprivileged tasks. Additional write protection can be used to prevent malicious or unintended modification of code when executing from Random Access Memory (RAM).

The code, data and stacks of inactive tasks are protected by dynamically reconfiguring the MPU during scheduling. This strict isolation prevents data corruption of inactive tasks,

Table 2.1: MPU access permissions [5]

AP[2:1] value	Access permission
0b00	Read/write by privileged code only.
0b01	Read/write by any privilege level.
0b10	Read only by privileged code only.
0b11	Read only by any privilege level.

e.g. caused by stack overflow or invalid pointer usage. Section 3.1 looks at implementation details of RTOS projects using this technique.

Logging these access violations during development can also help to easily uncover otherwise hard-to-track bugs.

2.1.1 MPU limitations

Despite offering a foundation for secure embedded software, advanced security architectures cannot be realized with an MPU alone. The following shortcomings demonstrate the need for additional hardware-enabled security measures:

- The number of non-overlapping memory regions is implementation defined and can vary between 0 and 8 for each security mode. This limit can be reached early with complex memory maps and fine grained configurations.
- Privileged code has full access to the entire memory, resulting in the RTOS kernel and all interrupt handlers to be part of the trusted code base. The addition of *TrustZone* solves this issue by adding another security domain.
- The MPU is located between the *ARM*-core and the system bus. For this reason, memory accesses can only be monitored when they are initiated by the core. As illustrated by Figure 2.1, additional components integrated into the SoC are not monitored and have no limitations when accessing the memory over the system bus [14, Ch. 5]. Unprivileged software could also use a Direct Memory Access (DMA) controller to modify or compromise secret memory areas [30]. Moreover, accesses from an external hardware debugger are not restricted. No selective debug permission can be configured, it can only be globally disabled with the control registers (e.g. *DBGEN*).

2.2 ARM Cortex-M TrustZone

Beginning with the *ARMv8-M*-Architecture, the idea of splitting software execution on a single processor core into two distinct security domains was implemented. Previously, specialized security modules or dedicated processor cores had to be integrated into the SoC to enable a secure execution environment. Complex interfaces and incompatible solutions from different vendors resulted in difficult development and a steep learning curve for beginners. Additionally, designing and integrating those security modules into a SoC resulted in higher cost, larger footprint and decreased energy efficiency.

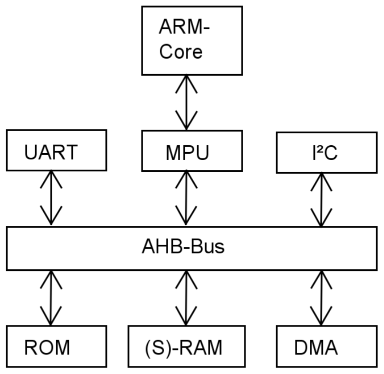

Figure 2.1: Simplified SoC architecture

With the introduction of the *Cortex-M Security Extensions (CMSE)*, mostly called *TrustZone*, an attempt was made to include a unified security architecture into the core. Because of the stricter requirements regarding energy efficiency, complexity and interrupt latency, the already established *TrustZone* feature of *Cortex-A* processors could not be reused. Section 2.3.1 highlights the differences between the architectures.

This new domain was added by extending the present privileged (handler) and unprivileged (thread) execution modes orthogonally. Figure 2.2 illustrates the relation between those four processor states. Switching between modes is fully hardware enabled and does not rely on hypervisor code, the memory address defines the permission instead. Section 2.2.3 gives more details about the supported transitions.

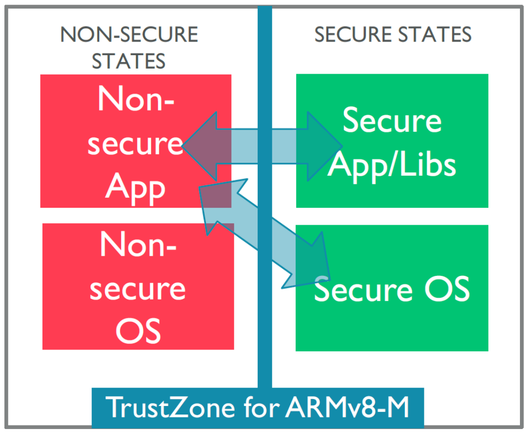

Figure 2.2: *ARMv8-M* execution modes [11, p. 5]

The memory is partitioned into secure and non-secure sections by software at startup. When the processor is executing in non-secure mode, only those memory sections can be accessed. During secure execution on the other hand, the entire memory is available. The special Non-Secure Callable (NSC) memory attribute can be used to provide entry functions for the non-secure software. Section 2.2.3 describes this process in more detail.

In contrast to the MPU, the security information is also propagated to devices on the bus, enabling protection of critical memory from bus members or debugger intrusion.

2.2.1 Attribution Units

Assignment of the security state to memory regions is done through the Secure Attribution Unit (SAU) and the optional Implementation Defined Attribution Unit (IDAU). The SAU is an integrated module of the processor core and can provide 0, 4 or 8 distinct regions with a resolution of 32 byte.

Memory which is not covered by any section, is considered secure when the SAU is enabled. Sections are used to carve out non-secure addresses. The same approach is used for peripherals, which are accessed through memory mapped registers.

Instruction execution and data accesses are checked against the SAU attribution. If the current security state of the processor does not allow the operation, a *SecureFault* is triggered.

The included SAU might not provide enough flexibility for some systems. For this reason, an interface for external attribution was designed into the architecture. The address of the current access is passed to the IDAU. The resulting information about the security state from the view of this external component is returned to special inputs at the core.

The complexity of this external logic is up to the chip manufacturer. It could be a constant lookup-table or a sophisticated configurable design.

The output of the SAU and IDAU are compared. The more restrictive of the two values is used. Figure 2.3 illustrates the address attribution.

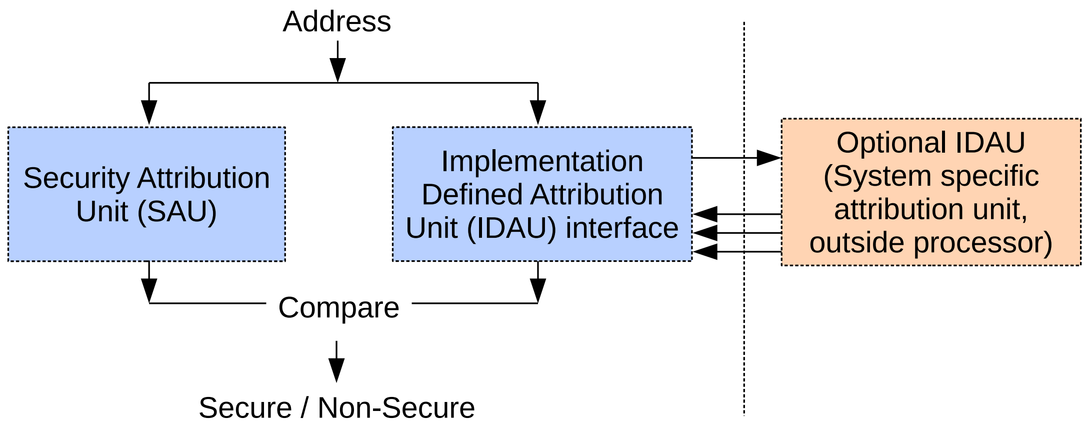
Address
Security Attribution Unit (SAU)
Implementation Defined Attribution Unit (IDAU) interface
Optional IDAU (System specific attribution unit, outside processor)
Compare
Secure / Non-Secure

Figure 2.3: Security attribution by SAU and IDAU [35, p. 7]

2.2.2 Banked system components

The following components are banked between security states and are switched automatically when changing the execution mode [35]:

- Stack pointers (Main Stack Pointer (MSP) and Process Stack Pointer (PSP))
- MPU (if implemented in the specific SoC)
- SysTick timer
- Various control registers
- Vector Table Offset Register (VTOR) (resulting in banked vector tables)

In addition to peripherals, some exceptions are also banked. Table 2.2 lists all system exceptions and states the banking configuration.

Table 2.2: List of banked exceptions [4]

Exception	Banked
Reset	No
HardFault	Yes
NMI	No
MemManageFault	Yes
BusFault	No
UsageFault	Yes
SecureFault	No
SVCall	Yes
DebugMonitor	No
PendSV	Yes
SysTick	Yes
External interrupts	No

Having duplicates of those components, especially *SVCall* and *SysTick*, opens many interesting use cases. For example, it is possible to run a separated RTOS in each security state with different scheduling, developed as independent projects. This application can already be classified as a simple virtualization technique.

The secure domain can also explicitly access the non-secure versions of all registers. This elevates the privilege of the secure domain above normal code.

2.2.3 Security domain transitions

One possibility to change the security state is by directly calling a function from memory of the other state. Additionally, higher privileged interrupts can also cause a spontaneous transition. These options might look at first like the cause of serious security issues, but this section describes the complex measures to prevent exploitations of this mechanism.

The base functionality is partially enabled by core hardware, but some additional assembler instructions had to be introduced. When using the *C*-programming language, it is toolchain and compiler dependent, which intrinsics have to be used. The syntax, which is used in the remainder of the document, targets the *Keil* compiler.

As illustrated in Figure 2.4, the *TrustZone* system is only concerned with horizontal transition which cross the security domain. Changes in the privilege mode are only relevant for the MPU.

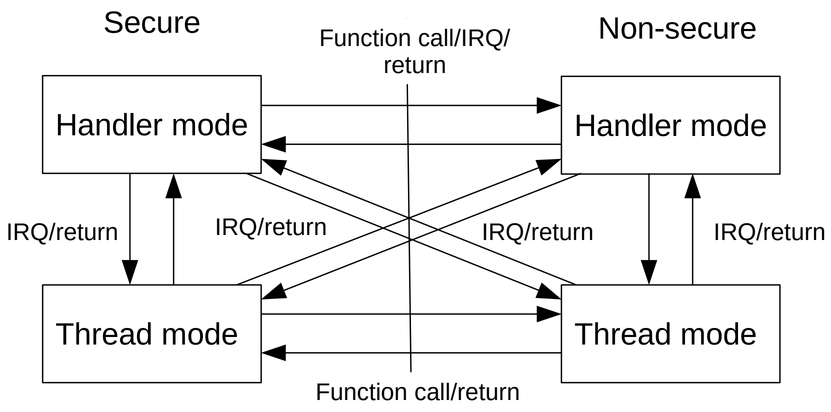

Figure 2.4: *ARMv8-M* domain transitions [35, p. 10]

Secure function call

One of the basic use cases is to provide a secure API which can be called from a non-secure application [36]. This is done by a simple function call when using the *C* language, but requires additional support from the toolchain.

A change to the secure state is only valid, when the first instruction which is fetched from Non-Secure Callable (NSC) memory, is the *Secure Gateway (SG)* instruction, otherwise a SecureFault is triggered. The SG instruction triggers switching of the security state and all banked registers. Subsequent instructions are executed in the secure domain and have full system access, without the need of a software hypervisor or management code.

Figure 2.5 illustrates the software flow which is generated by the toolchain: Secure API functions have to be decorated with the *cmse_nonsecure_entry* attribute. The linker does not create direct branches to these functions. Instead, function stubs called *secure veneers* are created and linked to initiate the state transition.

Creating additional veneers, instead of directly including the SG instruction in the secure function, prevents a possible security hole: Having the bit-pattern of the SG instruction in a data structure marked as NSC can open a door for arbitrary code injection. Even when only the Read-Only (RO) memory is marked as NSC, constant data and string literals could create malicious code by accident.

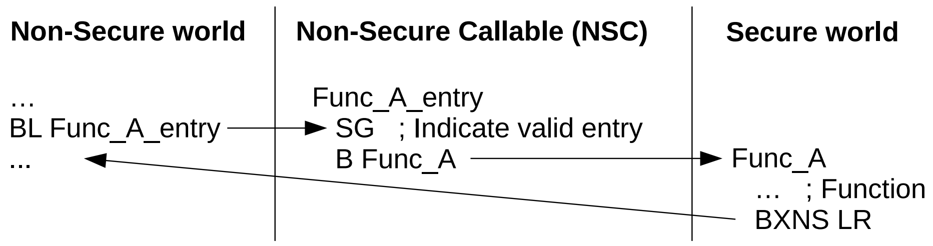

Figure 2.5: *ARMv8-M* secure veneer software flow [35, p. 9]

After the execution of the SG instruction, the internal state of the processor is switched to secure execution. For normal calls, the privilege level of the called secure code is derived from the secure banked *CONTROL.nPRIV* field. This prevents a privilege escalation when a non-secure, privileged code calls a secure API which should not have full access rights.

All banked registers, including stack pointers, are changed to their secure version. Non-secure registers can still be accessed by explicitly using aliased addresses.

After executing the secure function, a special branch instruction (BXNS) is executed to return to normal execution [4, p. 60]. This instruction has to be used explicitly to avoid information leakage of unintended mode switches, because additional clean-up has to be performed beforehand: General purpose and floating-point registers still might contain sensitive data which is not cleared implicitly. It is the responsibility of the toolchain or assembler programmer to overwrite those values manually. The use case might also require signalling of an imminent mode switch to external peripherals.

Non-secure callback

A use case might require calling a non-secure function from the secure domain. An example might be processing information from a secure interrupt by using non-secure functions.

While calling a non-secure function is an easy job, returning to the correct secure location has to be ensured. As the secure code is most likely compiled before the non-secure application, function pointers need to be passed and stored in the secure domain. These pointers have to be decorated with the *cmse_nonsecure_call* attribute to ensure that the toolchain clears secure registers and uses the *Branch and Exchange Non-secure (BXNS)* instruction for the call. Crossing the security domain with a normal *branch* instruction causes a *SecureFault* to prevent an unintended switch without clearing confidential information.

When the state switch is triggered, the secure return address, as well as parts of the Interrupt Program Status Register (IPSR) and *CONTROL* registers are automatically stacked to the currently active secure stack. A special value, called *FNC_RETURN* (0xFEFFFFFF), is written to the Link Register (LR).

When the non-secure code uses this value as return address to the secure code, automatic unstacking of the true return address is performed and the execution state is changed. This sequence keeps the return address secret and does not rely on the non-secure domain to return to the correct address. Figure 2.6 illustrates this sequence.

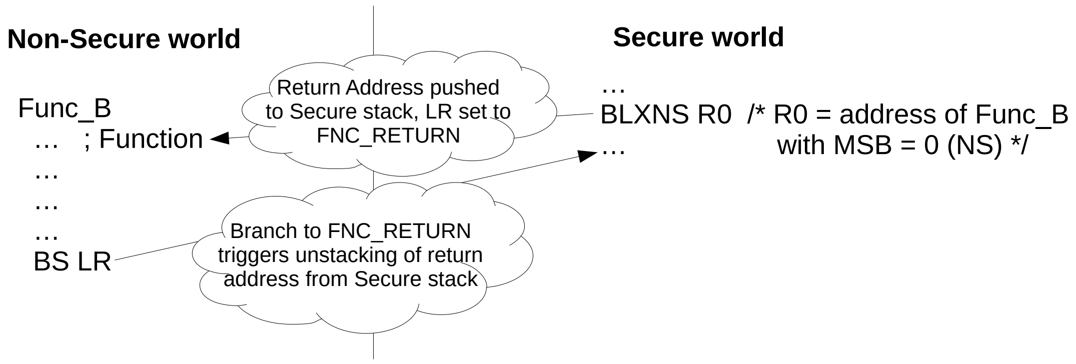
Non-Secure world
Secure world
Func_B
... ; Function
...
...
...
BS LR
Return Address pushed to Secure stack, LR set to FNC_RETURN
Branch to FNC_RETURN triggers unstacking of return address from Secure stack
...
BLXNS R0 /* R0 = address of Func_B with MSB = 0 (NS) */
...

Figure 2.6: Non-secure call-back sequence [35, p. 9]

Interrupts

The target domain of interrupts can be configured with the Interrupt Target Non-secure (NVIC_ITNS) register by secure software. The exception handler is fetched from the defined version of banked vector tables to execute the correct function. This mechanism can produce state transitions in both directions.

Which transition happened, is indicated by the *EXC_RETURN* value in the LR. It contains one bit for the source (S, bit [6]) and for the destination (ES, bit [0]) security state. Table 2.3 lists all fields of the *EXC_RETURN* identifier.

Table 2.3: *EXC_RETURN* field description

Bit number	Name	Value	
6	S	**0**	Non-secure code origin
		1	Secure code origin
5	DCRS	**0**	Callee-saved registers not stacked
		1	Default stacking
4	FType	**0**	Extended floating-point frame
		1	Standard frame
3	Mode	**0**	Handler mode origin
		1	Thread mode origin
2	SPSEL	**0**	Frame on main stack
		1	Frame on process stack
1	Reserved		
0	ES	**0**	Non-secure exception target
		1	Secure exception target

The simpler case, a transition from non-secure code to a secure interrupt, does not differ from the basic exception mechanism when staying in the same domain: The exception frame is pushed to the active stack, execution mode changes to handler mode and the exception handler is executed when no higher priority interrupt occurs. The only addition in this instance is the changed security mode. Table 2.4 shows the basic exception stack when no floating-point unit is used.

Table 2.4: Basic exception stack frame

Offset		
0x20		← original SP
0x1C	xPSR	
0x18	PC	
0x14	LR	
0x10	R12	
0x0C	R3	
0x08	R2	
0x04	R1	
0x00	R0	← new SP

Table 2.5: Extended exception stack frame

	Offset	Register	
Basic frame	0x48		← original SP
	0x44	xPSR	
	0x40	PC	
	0x3C	LR	
	0x38	R12	
	0x34	R3	
	0x30	R2	
	0x2C	R1	
	0x28	R0	
Additional state context	0x24	R11	
	0x20	R10	
	0x1C	R9	
	0x18	R8	
	0x14	R7	
	0x10	R6	
	0x0C	R5	
	0x08	R4	
	0x04	Reserved	
	0x00	Integrity signature	← new SP

Interrupting secure code by a non-secure interrupt results in additionally stacking of the callee-saved registers. This is done, because non-secure code cannot be trusted to restore the correct values on exception return. Further on, those registers are cleared automatically to prevent information leakage. Table 2.5 lists the contents of this additional state context.

2.2.4 Debug protection

Security attribution is performed on bus level. This allows for different debug permissions based on the security domain. Debugging can generally be classified into two categories:

1. **Invasive debug:** Conditions for breaking the execution flow by the integrated debug hardware can be configured externally. Examples are hardware breakpoints on specific addresses or ranges, and watches on certain register values. The core enters the debug state when a condition is met, and stops subsequent instruction fetches. The debugger can inject instructions or continue execution after collecting detailed data.

2. **Non-invasive debug:** The goal of non-invasive debugging is to monitor internal states and memory accesses, without interruption the execution. This can be used for

profiling the execution sequence of an application, where high real-time requirements have to be met (e.g. engine controller). Suspending code execution would completely break the functionality of the underlying technical process.

Both debugging types can be configured independently for each security domain. Additionally, reading of memory by the debugger can be restricted.

Disabling secure debug produces the following effects for an external debugger [31]:

- If an instruction step targets secure state, the function executes uninterrupted. Control to the debugger is not returned until the secure state is left again.
- Pausing the execution has no effect when secure code is executed. Debug state is entered only once a non-secure instruction is fetched.
- Memory and hardware breakpoints are ignored when they target secure memory.
- Secure configuration registers and the secure versions of banked registers cannot be read.
- Secure instructions and memory accesses are not traced by non-invasive debug.

Debugging can only be disabled for complete execution domains. Using this feature to enable IP protection for secure APIs, would also prevent third party developers from debugging their own secure code.

2.3 ARM Cortex-A

The target platform of this work is the *Cortex-M Series* of embedded processor cores. Despite having fundamentally different features and target applications, a lot of literature does make no explicit declaration of the used variant. To make matters even worse, the term *TrustZone* is also shared to the new Security Extension of *Cortex-M*, although the underlying functionality is vastly different. As a rule of thumb, when no explicit declaration is made, the author most likely addresses the more common version for the *Cortex-A* series. On the contrary, this document focuses on embedded *Cortex-M* processors.

Many related concepts of this work, like sandboxing, virtualization and memory protection, stem from the advanced field of *Cortex-A* processors. The following section gives an introduction and highlights the limitations when moving those concepts to an embedded architecture.

Ranging from low-performance appliances, over mobile devices up to enterprise server clusters: *Cortex-A* processors can be found in many application domains. The most relevant differences for this project are interrupt handling, memory management and execution modes [33].

- **Interrupts:** *Cortex-M* guarantees short interrupt latency by handling all stack and branch operations directly in hardware. Application processors on the other hand, may require multiple stages of function calls and mode switches until the correct interrupt handler is executed.

- **Memory management:** By providing a Memory Management Unit (MMU), *Cortex-A* processors enable hardware support for virtual memory and allow the execution of rich operating systems like *Linux* [20].
- **Execution modes:** In addition to the privileged and unprivileged execution modes of embedded processors, *Cortex-A* provides two additional exception levels. The *Hypervisor* and *Secure Monitor* modes add additional privilege levels. More details are given in Section 2.3.1.
- **Virtualization:** With the additional *Hypervisor*-mode, a two-stage address translation allows the execution of multiple, isolated operating systems in the non-secure world. A wide range of available hypervisors offer a large variety of different security features.

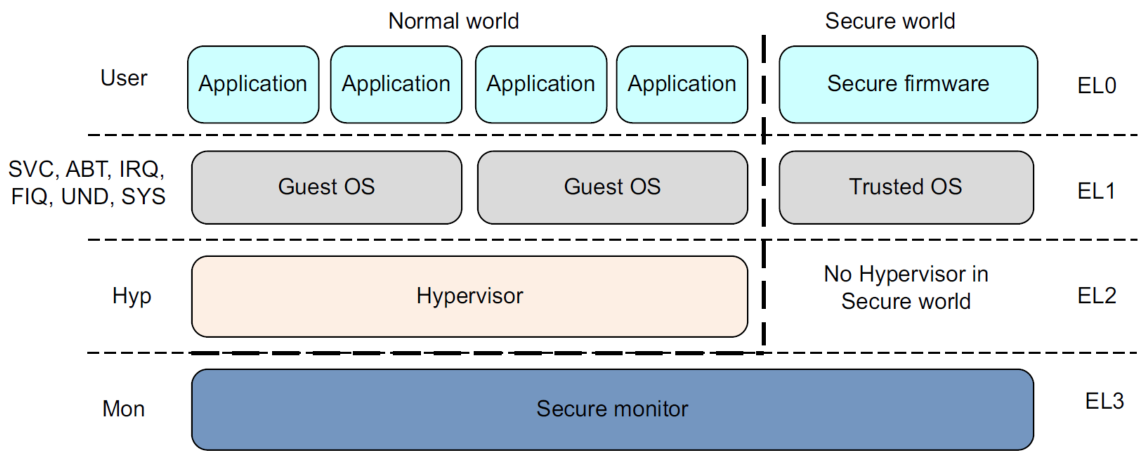

Figure 2.7: *Cortex-A* execution domains [20]

2.3.1 ARM Cortex-A TrustZone

In contrast to the *Security Extension* of *Cortex-M* processors discussed in Section 2.2, switching to the secure domain cannot be done directly with a simple function call. The entry to the secure-world must always be performed with a Secure Monitor Call (SMC) instruction.

Direct access to secure memory is restricted with additional hardware logic in the bus fabric. Every transfer includes a dedicated bit to mark the security status. The slave interface compares the state with the internal configuration and blocks the access if invalid.

This additional bit can be viewed as an extended width of the address bus. Resolution of virtual addresses takes this into account, resulting in an aliased memory space with two *"virtual"* MMUs. This flag is also regarded for cache architectures to avoid information leakage.

The *Secure Monitor* is responsible for forwarding the call to the correct secure-function based on the supplied parameters, or trigger more advanced events when a dedicated operating system is available in the secure domain. Handling of this RPCs is implementation dependent, but API standards (e.g. *ARM SMC calling conventions* [25] or *GlobalPlatform TEE API* [28]) were introduced to provide better portability and compatibility between multiple manufacturers.

Many security critical functions have to be performed by this software component, which is often provided by the SoC-Manufacturer. Events from the past have shown, that security issues in this central part can compromise the entire system [9][22].

Another limitation of the *TrustZone* is the missing hypervisor mode in the secure world. This circumstance prevents isolation of multiple unrelated trusted modules with the build-in virtualization techniques [10].

2.4 Point of Sale Technology

With the increasing acceptance of cashless payments by customers, most merchants have an integrated point of sale system. The following gives a short introduction to the interacting components of such a system. Because the application of this work is focused on payment terminals, they are described in more detail.

The visible payment terminal is just a part of the overall system. It usually provides a card reader to authenticate the customer, a display for status information and a keypad for Personal Identification Number (PIN) entry. An additional contact-less interface, using NFC technology, is also included in most recent terminals. External interfaces, either wired or wireless, connect this hardware to the rest of the payment system.

Based on the card type and state, different sequences for accepting the transaction are possible. When the card is inserted into the card interface, the PIN is required for authentication. In case of an online transaction, contact to a remote payment processor is established to check the availability of funds. The received status is returned and transactions can be reported to the attached warehouse system when the sale was completed.

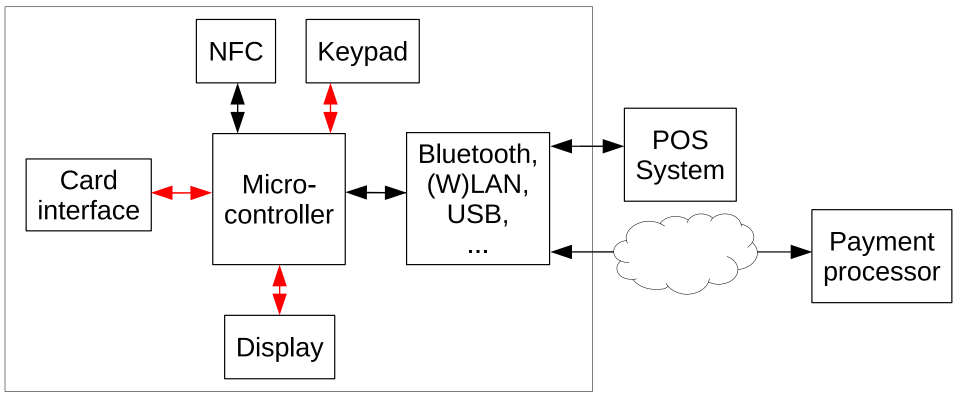

Figure 2.8: Simplified POS architecture

This simplified view already illustrates the serious requirements on security. The complete system can be compromised by successfully attacking a single component. For example, all red arrows of Figure 2.8 might handle the PIN.

A common attack scenario is compromising the PIN and Primary Account Number (PAN) data. This information is sufficient to create forged transactions from the victims account. Sensitive data might be stored by malicious components inside the firmware. It is later send to a third-party, which withdraws money in cash, initiates transactions to services or buys goods. Typically, hard to trace products are bought and later resold (e.g. gift cards, crypto currencies).

As a first line of defence, tamper switches and conductive meshes around the circuit detect modifications. Once those precautions are triggered, the device should no longer operate normally and lock down sensitive data. This should prevent the unnoticed installation of modified firmware on the terminal Authenticated boot images and encrypted secret data is further used to restrict the possible influences of a successful attack.

From the many possible attack vectors, this design focuses on the memory and firmware security aspects. One goal of this project, is to increase the resilience of the firmware against injection of malicious code. The context manager helps with this problem by reducing the Trusted Code Base (TCB) which is in contact with sensitive information. This prevents a fault in unrelated components, e.g. GUI from being used to compromise the entire system.

2.4.1 PCI Security Standard

The Payment Card Industry (PCI) Security Standard tries to define minimum requirements for products of the payment industry. Ranging from card production to network and software security, the main goal is to protect the data of the cardholder.

Payment terminals are designed to prevent tampering on the hard- and software. This should make adding of backdoors or keyloggers impossible. Nevertheless, there are always unpreventable weak spots in a system.

Practical attacks are performed on the device, with the goal of successfully implanting a keylogger. The time requirement and the needed expertise for a successful attack is rated by the standardisation. This quantifies the probability and feasibility of a theoretical attack vector. A minimum rating has to be reached for passing the certification process.

There are also static requirements defined: Triggering the tamper switches must lock or delete the internal keys within a certain time frame.

Another regulation is the mandatory audit of security sensitive firmware, which should evaluate the design concept and prevent critical bugs in the implementation. The firmware itself has to perform authentication during boot and regular integrity checks during operation.

Relaxed rules are mandated for non-critical code which is isolated from the secure firmware. The *TrustZone* and context manager could be used to exclude large parts of the non-secure application from these strict rules. Updating of non-critical code could be performed without needing a recurring audit.

Chapter 3

Related Work

The concept of task isolation is already well established in the world of embedded operating systems. With projects targeting the ARM Cortex-M platform, the basic operation always follows the same principle, which can be partially applied to the use case of this project: During task scheduling, the memory and independent stacks of the inactive tasks are disabled with the MPU (as described in chapter 2.1).

This mechanism provides basic isolation of unprivileged tasks to prevent memory corruption and leakage of sensitive information. It can also be used to protect the operating system resources and privileged tasks from side effects caused by a corrupted module [16].

Having the support from an MMU and a rich operating system, software for *Cortex-A* processors is generally not concerned with this low-level task/thread management. The RTOS kernel on the other hand uses the *TrustZone* to provide secure and trusted services to the application. Another advanced hardware feature is the support for hardware-virtualization, enabling the concurrent execution of multiple operating systems with strong memory separation [20].

The following sections introduce example products from different application domains and highlight the differences. The gained knowledge of these architectures is combined to produce a design which leverages the best aspects of tried and tested solutions.

Evaluation is divided into three main categories:

1. **Embedded (RT) operating systems:** Many embedded operating systems provide support for basic isolation of tasks by using the MPU, which was already introduced in Section 2.1. Examples of major distributions are evaluated to re-use the underlying concepts in this project.

2. **Cortex-A TrustZone related:** Enabling an isolated and trusted execution environment is an established feature for application processors. With the many use cases of mobile devices which require software security, advanced architectures were developed. This section looks into those applications and derives low-level functions for use in embedded systems.

3. **ARM Platform Security Architecture:** With the introduction of the security extension to the *Cortex-M* platform, *ARM* attempts to increase device security by providing reference architectures and implementations of core components. Section 3.3 focuses on the *Secure Partition Manager* of this project.

3.1 Embedded operating systems

Looking at the implementation of embedded RTOS projects acts as reference for task switching and memory protection concepts.

3.1.1 embOS

Developed as commercial product by SEGGER, embOS is a complete RTOS solution. It provides advanced features like events, mailboxes and preemptive task scheduling [7].

The base version does not provide memory protection and distinction between privileged and unprivileged tasks.

To use the MPU features, the *embOS-MPU* extension is required. Tasks are started in privileged execution mode, perform initialization for themselves and switch to unprivileged state, where they are restricted to their protected memory and communicate with the kernel using an API. Access to device drivers is also limited to an indirect call to the kernel.

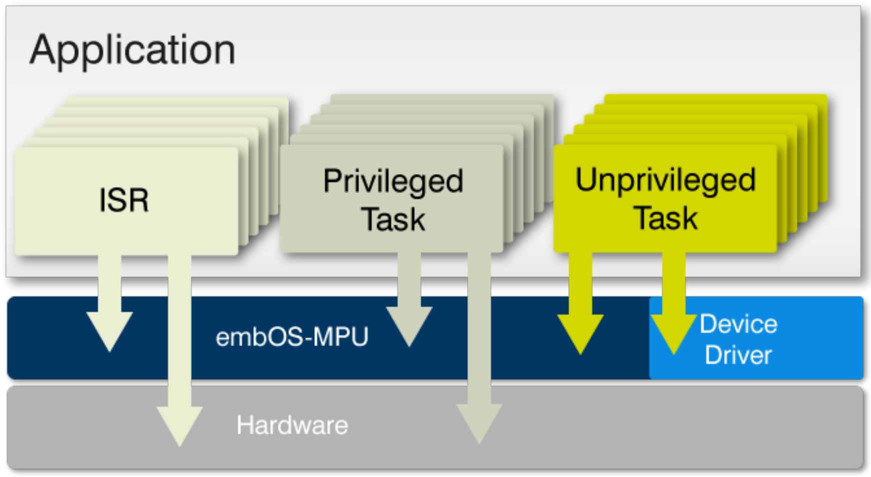

Figure 3.1: embOS task handling [23]

All privileged tasks and interrupt handlers have full access to the complete memory, an additional isolation for this code is not performed. Additionally, access to hardware and kernel memory can be explicitly granted to individual unprivileged tasks with a whitelist approach.

Because of the commercial nature of this product and the associated high license cost, no evaluation of the internal implementation can be made by the author. Support for the *ARMv8-M* architecture is advertised but the functional range is not revealed in detail [24].

3.1.2 FreeRTOS-MPU

The *FreeRTOS* project is developed for more than 15 years and supports a wide range of officially supported architectures, including *x86*, *AVR* and *ARM* [18]. Managed by Amazon Web Services (AWS), it stayed completely open-source and is counted among the most used RTOSs in the embedded market [6]. Only the relevant security features for this project are mentioned in the following.

For *ARM* platforms supporting an MPU, the kernel source and data are mapped to an exclusive memory section, which are protected from unprivileged access at start-up. Individual tasks can be created with up to 3 protected memory sections [29]. During scheduling of the task, those sections are activated by the MPU and protect other data from corruption due to faulty write accesses. The exclusively owned stack is also bounded by a MPU section to prevent against the severe side effects of stack overflows.

Code and constant data of unprivileged tasks is configured as read-only by default. This prevents accidental corruption by software faults and code injections from an adversary. Hiding of unprivileged code is not supported by default, but could be implemented with the following steps:

- Define an exclusive memory section for the task code in the linker script.
- Place the code in this section with compiler attributes or linker settings.
- Read the linker symbols defining the address range.
- Specify the code section when creating the restricted task with the *FreeRTOS* API. It will be disabled when the owning task is not currently executing.

The open-source nature of the project allows for a detailed look into the scheduler and task switching operation. Using the Pended Service Call (PendSV) exception, the kernel is triggered to perform a task switch. The general-purpose registers (including Floating-Point Unit (FPU) registers when used), *CONTROL* register and the return address are pushed to the stack of the interrupted task. Subsequently, a scheduling algorithm selects the task with the highest priority. The PSP address is exchanged with the value of the private stack of the next task. Restoring the saved registers finalizes the task switch.

Using the *TrustZone* for kernel isolation and the changed programmer's model of the *ARMv8-M* MPU are not yet supported by this project.

3.1.3 mbed RTOS - uVisor

Managed directly by *ARM*, this open-source RTOS targets IoT solutions. In addition to the kernel, it includes device libraries and drivers aimed for communicating with a server infrastructure. This cloud extension can be used for secure communication, as well as updating the application software [1].

uVisor

Device security is enforced by the optional *uVisor* security kernel. It protects the operating system from unauthorized modifications and enforces isolation between application tasks and their resources. An attacker compromising one task, can no longer influence the complete system. Security critical components, e.g. firmware update or encryption functions, are no longer vulnerable because of untrusted application code.

When creating a new task, an Access Control List (ACL) is defined with provided macros. The task can only access memory addresses defined public and those from the private ACL. The kernel implicitly reserves a section for a private stack.

Although it is a complex and well-tested system, *ARM* considers *uVisor* to be an technology preview with possibly incomplete implementations [2]. With the current version of *Mbed*, *uVisor* is deprecated and not the focus of further development. It is advised to switch to the *Secure Partition Manager* introduced in Section 3.3.

Unprivileged interrupts

The *uVisor* software provides a functionality for de-privileged interrupt hooks. Functions are registered by tasks on a first-come-first-serve basis.

The interrupt vector is swapped out during initialization of the kernel. All interrupts are reconfigured to target a generic wrapper, which triggers a privileged Supervisor Call (SVC). A dedicated gateway function retrieves the original interrupt number, activates the associated context, clears the general-purpose registers and jumps to the interrupt handler. Execution privilege level is decreased beforehand and execution will resume at the entry wrapper after finishing the interrupt handler. To resume normal execution, another SVC is made to restore the stack and permissions of the interrupted context.

Remote procedure calls

Secure API calls cannot be made directly. A task providing API functions has to register them for public usage explicitly. An entry function is placed in globally readable memory, which triggers switching of the active context and a call to the actual function.

Public functions are callable from every context. The implementation has to query the task identifier and validate the source of the access itself. No support for access restrictions is provided by the kernel.

3.1.4 ProvenCore-M

The commercial *ProvenCore* operating system focuses on enabling the development of secure devices, especially for the IoT market. The customer should only concentrate on application development without needing the knowledge of advanced security concepts [21].

Deployed applications are isolated from each other to provide security and stability. The kernel is already formally proven, which simplifies a required certification process.

Illustrated by Figure 3.2, multiple partitioning concepts are possible:

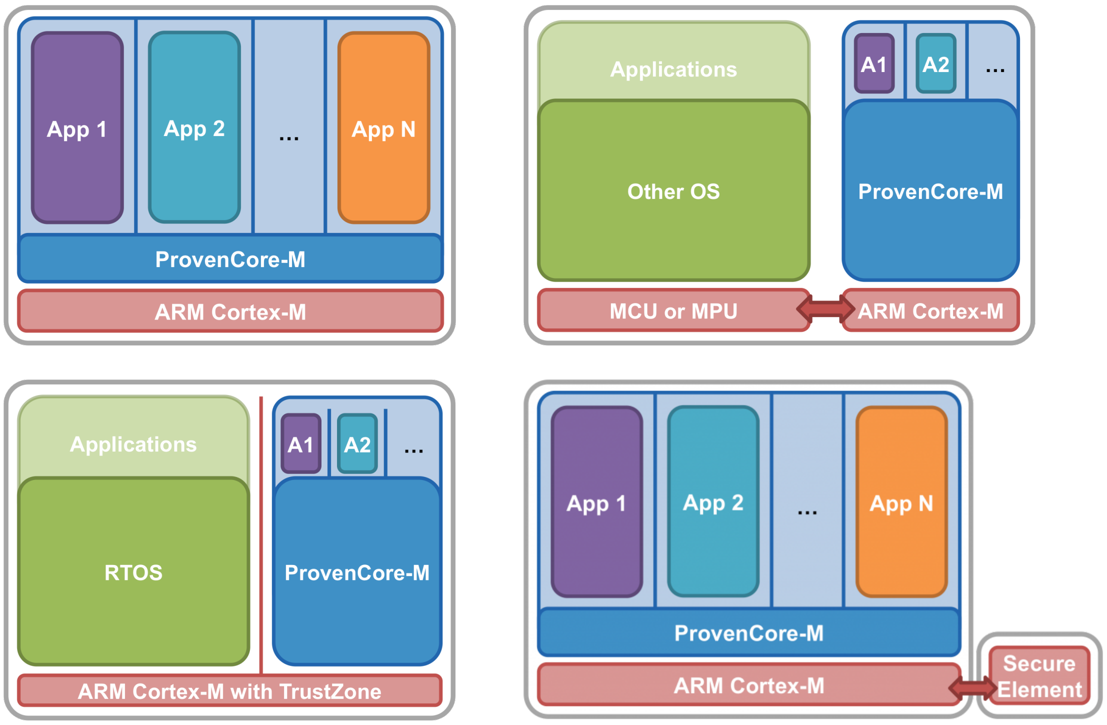

Figure 3.2: Partitioning concepts of *ProvenCore-M* [21]

- Application isolation can be realized on generic *ARM Cortex-M* microcontrollers without special security features. This can be utilized for secure storage or protecting privileged drivers.
- When a security coprocessor is present, *ProvenCore* can be deployed on it to manage the secure applications. Communication from the non-secure application, running on the main processor, is managed by the kernel.
- The target partitioning of this project is also advertised: A non-secure operating system manages the application on a *TrustZone*-enabled microcontroller. The build-in functionality protects the secure applications from it, with further isolation between the applications provided by the security kernel.
- Platforms with an integrated *Secure Element* are also supported to provide secure storage and cryptographic functions. Access rules are enforced by the kernel to prevent leakage of secure data.

Because of the commercial nature of this project, all statements are based purely on the marketing information from the company. Detailed description about how the presented concepts are implemented is not published.

3.2 Cortex-A TrustZone examples

Basic memory protection is handled in *Cortex-A* processors with a dedicated MMU controlled by a rich operating system. Every process performs operations on virtual memory addresses which are independent from the actual physical locations. Moreover, processes have only unprivileged access rights and communicate with peripherals over system calls provided by the operating system.

This section is not concerned with the basic principle of virtual memory management. Instead, it gives an overview of using the *Cortex-A* specific *TrustZone* for security critical functions. The main characteristics of example projects are summarized to show the practical application of this technology. Section 2.3.1 already introduced the main features of the underlying platform.

3.2.1 Android

The Android kernel is executed entirely in non-secure world. Secure system services are decoupled from the rich operating system and placed in the secure execution environment.

Access to secure services is established with SMC calls, which trigger the hypervisor and a secure kernel. This secure kernel is an independent operating system located in the secure domain. Individual services are managed as tasks and isolated with the MMU.

The feature set of trusted services is platform specific and often not published, but the *GlobalPlatform API specification*[28] acts as standard for *Android* devices. The SoC manufacturer develops the secure kernel (e.g. QSEE from *Qualcomm* [22]) and integrates secure services for the chip integrator.

The following lists possible applications for secure services:

- Secure storage: Certificates and manufacturer keys are stored in protected Read-Only Memory (ROM). Additionally, session keys and secret runtime data can be managed in secure RAM sections.

- Secure boot: Execution starts in the secure domain to perform verification of the system kernel. Faulty configurations are reset, and the system can be started to a reset/update mode. Periodic checks can also be performed to notice run-time modifications and non-typical behaviour.

- Digital Rights Management (DRM): Multimedia applications often rely on proprietary DRM functions for decoding the content. It is in the interest of the provider to protect encryption keys, algorithms and the decoded data from unauthorized accesses.

- Advanced authentication: Authentication with biometric features is a security critical process. Accessing raw data, e.g. fingerprint or face features, must only be allowed from secure code. Optimized reference data has to be protected from unprivileged accesses to avoid the possibility of injecting data from an attacker, or leaking details which could be used for forging a duplicate.

3.2.2 ANDIX OS

ANDIX OS is a security kernel developed as an open-source project aimed for research and educational purposes. The kernel is executed in the secure domain and manages the memory permissions and scheduling for trusted applications. Figure 3.3 illustrates all system components.

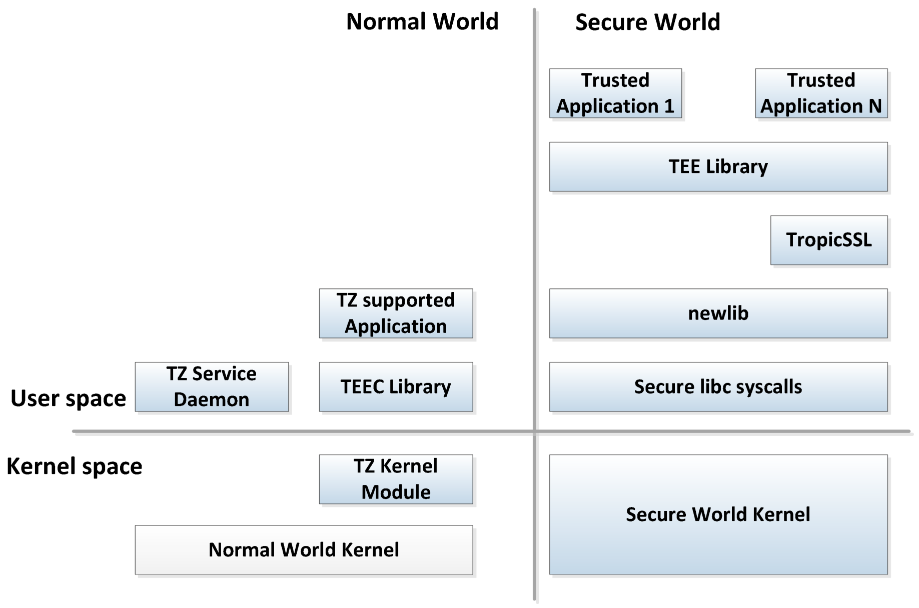

Figure 3.3: *ANDIX* system architecture [8, p. 32]

The non-secure world can in theory run any full featured operating system, but the needed kernel extensions are currently only ported to *Linux* and *Android*. This kernel extension copies the request data from the user-space to protected kernel memory to prevent subsequent changes by the unprivileged application task. The SMC instruction is then used to switch to the secure domain and trigger the *ANDIX* kernel.

Because the SMC instruction can only be executed in privileged execution mode, an additional user-space library is needed. This library implements the API functions defined by *GlobalPlatforms* [28] and forwards them to the non-secure kernel extension with a SVC instruction.

In the secure user-space domain, the *TEE Library* also provides a standardized API for communication to the secure kernel. Example functions include memory management, timing, trusted storage and cryptographic operations. Only a limited subset of the API is currently implemented by *ANDIX*.

The *TZ Service Daemon* is executed as application task in the non-secure user space. It regularly polls a specific memory location used by the secure world for non-secure call-backs. Those call-backs are predefined service functions which are complex, non-critical functions which are moved from the secure domain to reduce the TCB of the system. They also allow later addition of hardware drivers and utilities, e.g. a complete networking stack.

The central part of the kernel manages the isolation between trusted applications. This is achieved by changing the translation table of the MMU during context switching. The task can only access data which is already mapped to its virtual address space, physical addresses of other tasks or the kernel are protected.

3.3 ARM Platform Security Architecture

The proposed Platform Security Architecture (PSA), developed by *ARM*, is meant to be a solution to the security issues faced in the IoT market [19]. With an expected landscape of billions of connected devices from diverse manufacturers, the already mentioned security threads cannot be ignored.

With the high market share of *ARM* microcontrollers in this business segment, a common framework is applicable on a wide range of devices. *ARM* wants to develop a base layer for their platform to increase security and reliability by reducing the responsibility of the manufactures. The PSA supports developers by giving a guideline for the three important development phases:

- Examples of thread models are analysed for common use cases. They can be applied to the custom application or serve as starting point of a completely new analysis. Guidelines are published, to act as a reminder for known security considerations and new threads in this area.
- Based on the preceding analysis, the PSA is derived. It provides best-practice designs for common security components. Modules like trusted boot and firmware updates are discussed here. Designs for advanced scenarios for the complete device lifecycle are given. This ranges from secure manufacturing to provision the confidentiality of root-secrets and also includes details of the concrete firmware architecture (e.g. authentication, partitioning).
- Maybe the most effective measure to increase the security of a wide range of devices, is the provided reference implementation. Developed as open source, a complete base architecture is published with the Trusted Firmware-M (TF-M) project. The next section focuses on the features of this implementation.

3.3.1 Trusted Firmware-M

The first version of TF-M was released in March 2018. At the time of this writing, it is still under active development with missing features. The current roadmap predicts a complete implementation with the second quarter of 2019. This time frame will approximately correlate with the widespread commercial launch of *ARMv8-M* processors. Figure 3.4 illustrates the structure of the system components.

The following lists key points of the planed feature set:

- As root of trust, a secure bootloader with image upgrade functionality will be integrated. Authenticated booting of the firmware is ensured for secure and non-secure binaries.

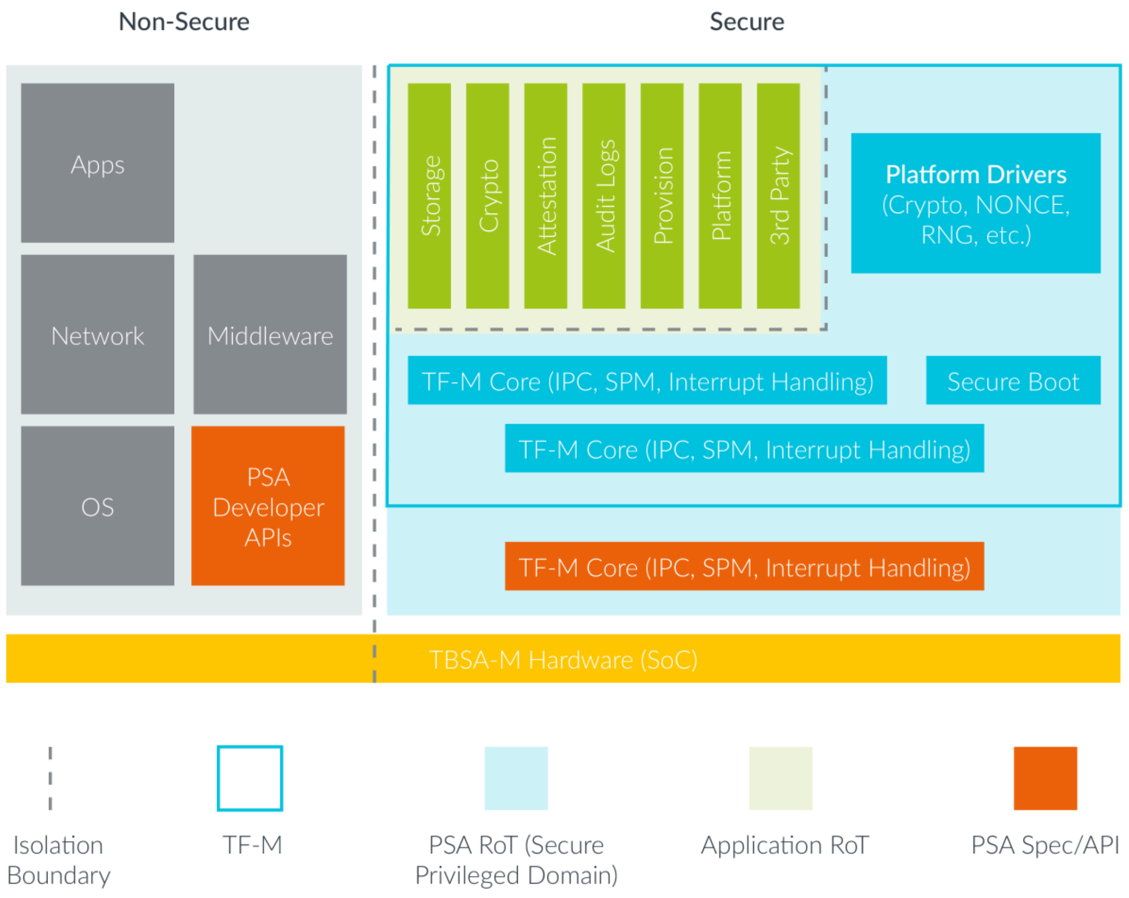

Figure 3.4: Structure of Trusted Firmware-M [19]

- The fundamental design already takes a non-secure and a secure processing environment into account. This can be realized with dual-core hardware, where both domains run on different processors. The main goal however, will be to use a single processor with the *TrustZone* extension for this separation.

- In addition to the basic separation, secure services will be placed into isolated partitions. This higher isolation level is also used to protect the secure core from malicious accesses from compromised services.

- A dedicated Inter-process communication (IPC) framework and standardized APIs are used for communication between distinct isolated secure partitions.

- The non-secure RTOS is completely decoupled from the secure *Trusted Firmware* components. The current implementation supports *RTX* as guest-OS, but other major RTOSs, like *Mbed OS* and *FreeRTOS* will also be supported in the future.

Because of the early stage of development, individual components of the project could not be re-purposed for solving the task of this work. Implementation for the secure partition manager will not be finished before the first quarter of 2019. No details about the expected overhead and latencies is published. This prevents a more detailed evaluation and comparison to other projects.

Nevertheless, it is an ambitious project with the potential of causing a lasting security increase in the IoT market.

3.4 Conclusion

By looking at similar concepts and evaluating already proven projects, new ideas and potential design choices are contributed to the solution of this work. This section looks at key points which will be considered for the final design.

The basic task management, which is used by most RTOS schedulers, should be applied to the context manager. Individual stacks are reserved for every context.

Instead of implementing privileged and unprivileged tasks like *embOS*, the *TrustZone* and firewall could be used for a similar concept. Direct access to the configuration registers of hardware components could be blocked for non-secure code. The separation of the resulting architecture would only allow secure libraries a direct hardware access. Non-secure application code must use device drivers implemented as secure API functions.

Kernel functions and data are explicitly placed in separated linker sections by the *FreeRTOS* project. This is done by manually decorating the source code with the *section* compiler attribute. The ranges of those automatically scaled regions are read back by the initialization code from linker-symbols. The MPU is then configured to block unprivileged accesses to those memory sections.

Explicit placing of code and data, by using compiler attributes and linker sections, will also be used by the context manager. Linker symbols should be used to synchronize the address ranges between linker script and configuration code.

The *mbed OS* provides the functionality for de-privileged exception handlers. Application code can implement interrupt service routines by the default way. The secure kernel changes the interrupt vector and adds gateway functions to lower the privilege level before executing the code from untrusted sources.

The feature descriptions of *ProvenCore-M* and *Trusted Firmware-M (TF-M)* show the possibility of isolated partitions inside the secure execution domain. Little information is given about the implementation, but the MPU is apparently used to enforce the permissions.

Disregarding the little information which has been gained from the *ProvenCore* project, an important point was raised: An advantage of this project is the formal verification and the provided support for certifications. Other projects might require extensive audits for large parts of the kernel.

It also raises concerns about including a full-featured RTOS in the secure domain altogether, because the large TCB makes a detailed verification impossible.

Chapter 4

Design

This chapter describes the design decisions based on the related work and the target use case. The resulting implementation has to satisfy the following requirements:

- **Context protection:** Protection against unauthorized accesses to secure code and data is already provided by the *ARM* security extension. The main goal of this project is the separation of individual contexts within the *TrustZone*.

  One requirement is to ensure the integrity of code and data in case of stack-overflow or erroneous memory operations of another context within the secure domain. Another aspect deals with IP protection: Arbitrary secure code must not be able to read the entire secure code-base and potentially dump it to an external peripheral. This prevents reverse engineering of proprietary software.

  Advanced attack scenarios, e.g. violating the *ARM Application Binary Interface (ABI)* by passing an incorrect return address or corrupting the callee-saved registers are discussed theoretically but are not fully covered by the implementation.

- **Small Trusted Code Base (TCB):** The context manager is a fundamental part of the overall system security. As the module becomes more complex, the probability of critical bugs increases. This constraint rules out the possibility to modify an existing RTOS-kernel for the target application. Including an entire RTOS would present a large software overhead, especially when the resource constraint environment for embedded systems is considered.

- **Influence on NS-domain:** Context isolation should only be implemented for the secure execution of the processor. Non-secure software should not be influenced by this addition. Especially when porting legacy projects to the new platform, the required modifications should be kept to a minimum. At the best case, only changes to security critical libraries and the build scripts are required.

- **Cross-domain calls:** Especially for complex projects, there must be no restrictions on available cross-domain calls. In connection with the previous requirement, a complex RPC-API should be avoided. The provided transitions of the security extensions should be used as primary entry point. This also leaves the basic security responsibility to the build-in hardware components. The context manager is only concerned when a reconfiguration in the secure domain is required.

4.1 Target hardware

At the time of this writing, no microcontrollers based on the *ARMv8-M* architecture are publicly available. The only options to develop code for the new security extensions are the following:

- ARM developed the *Musca-A Test Chip Board* as an official development platform. Availability of those boards is still limited and requires entry to an application process. No details about the acceptance criteria and cost are published.
- A complete virtual system can be emulated with the *Fast Models* provided by ARM. A licence for the so-called *Fixed Virtual Platform* for Cortex-M33 is available for a price of 500 dollars per year.
- A soft-core can be synthesized for a Field Programmable Gate Array (FPGA) target. Running a complex model requires potentially expensive hardware.

All public solutions had a high price compared to the offered features. The additional effort of organization and setup of the toolchain made them further unattractive. Fortunately, an internal prototype was provided by *NXP* for this project. It is a development board, used for implementation and testing of an upcoming product. Featuring a Cortex-M33 core with the optional security extension, and an additional bus-firewall, it was an ideal target for implementing the context manager. The firewall is described more detailed in the next section, as it plays an essential role for this design.

The core of the chip architecture consists of the Cortex-M33 clocked at 100 MHz, an AHB-lite matrix and 256 kilo-byte (kB) of RAM. External peripherals are supported by DMA enabled communication controllers (e.g. SPI, UART and I^{2}C). Execution on this controller is mainly RAM-based. The encrypted binary is loaded from external memory by a small bootloader located in ROM. With the help of a cryptographic coprocessor, the decrypted code and data is written directly to the RAM. With this approach, clear text code is never located in non-volatile memory.

Additional components of the SoC are cryptography blocks, one-time-programmable memories and an independent Digital Signal Processor (DSP). Those modules are included for another application and will not be actively used by this project.

4.1.1 AHB-Firewall

The proprietary AHB firewall was developed to enforce flexible memory permissions on bus level. It solves the all-or-nothing constraint for DMA masters: Access to individual components can be blocked on hardware-level with the bus matrix. The connection to the boot-ROM might be configured exclusively to the CPU, for example.

This static configuration cannot be changed later and does not allow distinct sections within a memory module. Moreover, the type of access is not considered for defining the permission.

The combination of a SAU and IDAU for security attribution is described in Section 2.2.1. To reduce the total gate count, no SAU regions were implemented in favour of an integrated IDAU into the firewall. Instead of defining a constant number of sections, security attribution is based on memory blocks. A distinct security setting can be configured with a resolution of 4 kB. The IDAU is responsible for the basic security attribution of the memory. In addition to that, configurable sections make a much more precise configuration possible. Table 4.1 lists the available configuration parameters for every section.

Configuration is partially predefined by hardware for the used processor: The different memory modules have a fixed number of available sections with static resolutions assigned to them. Dynamic configuration by software can only be used for moving those windows within the boundaries. An 8-bit mask is provided to split the section and apply the rules on non-contiguous memory ranges. For every section, the following configuration is possible:

- Debug can be explicitly disabled for the section, to override the global configuration.
- The permission of CPU accesses can be configured for read, write and execute. The secure and non-secure execution state is defined independently. Information about the current execution mode is directly received from a default interface of the ARM-core.
- Read and write operations can also be enabled for secure and non-secure DMA masters. Assignment of the security state of each member is done in software, with optional locking described in the next section.

Table 4.1: Firewall rules configuration

Field	Setting
Modification allowed	
Enable	
Debug	Forbid, Global configuration
Secure CPU	Read, write, execute
Non-secure CPU	Read, write, execute
Secure DMA	Read, write
Non-secure DMA	Read, write

When an AHB master initiates a transfer, the address is compared with the configured rules and permission of the component. In case an invalid access is observed, the firewall triggers a *Firewall-Interrupt* and cancels the bus operation.

Configuration locking

The configuration of the firewall is generally protected by itself: The address range of the memory-mapped registers should be covered by a section with restricted privileges. Write access is only granted to secure software as a first defence against unauthorized configuration changes.

An additional mechanism further protects some configurations from changes by secure software. All security critical registers contain additional *"Modification-allowed"* flag. Updating of values is monitored by an internal logic. Based on the setting, modifications can be blocked completely, or they can be allowed when they are more restrictive.

Updating of the configuration can be locked for the following registers:

- Debugging can be disabled globally from a single register. This configuration can be frozen until the next reset of the chip is performed.
- The access rules of every section are configured with associated registers as described previously. Modification can be controlled in three steps: All modification can be allowed, blocked or more restrictive settings are permitted.
- The assigned addresses are locked together with the rules register, to prevent bypassing of the firewall by modifying the effective range of the section.
- Security state of AHB masters is also a critical information. Once a master has performed all secure responsibilities (e.g. during start-up), it can be downgraded and locked to non-secure status.

4.2 Software partitioning

Ownership of the individual components is divided in the following way:

- Bootloader and context manager are provided by the chip manufacturer, possibly on an internal ROM. The responsibility of those components is to load the trusted firmware, set an initial configuration and provide the context isolation. This scenario lays the foundation of a Root-of-Trust (RoT) chain.
- Low-level drivers, in this example for SPI and Contactless Interface (CLIF), are also provided by the chip manufacturer. The customer can use sophisticated APIs instead of explicit register accesses. As this software may reveal a lot of information about the underlying hardware modules, protection against read-out and reverse engineering is in the interest of the manufacturer.
- The system integrator provides the core firmware for the payment terminal functionality. In this example, security critical payment libraries and hardware abstractions for a connected pin-pad are shown. Those components furthermore use the low-level drivers of the chip manufacturer.
- The enclosing application with the RTOS is isolated from security critical components by the *ARM Security Extension*. It can be supplied by the system integrator or by a final licensee. Function calls to secure libraries cross the barrier to the secure domain. Depending on the system architecture, additional functionality might be extracted from the non-secure application and placed in the secure domain.

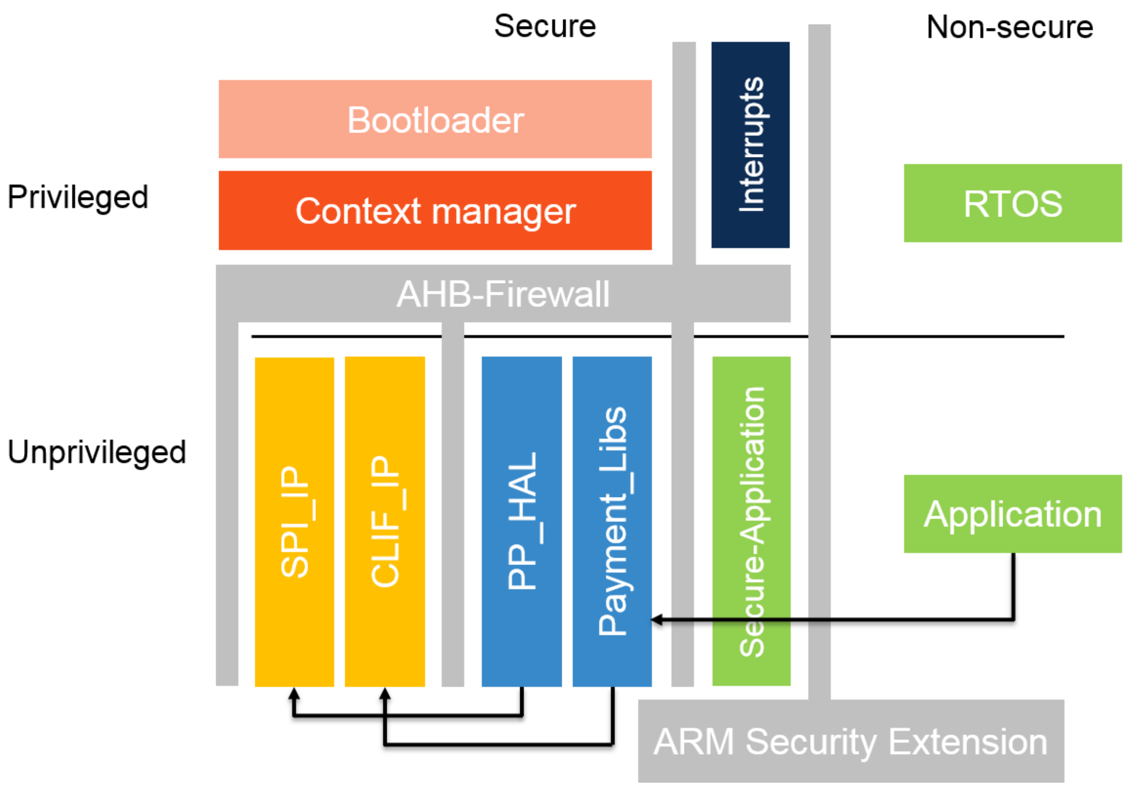

Figure 4.1: Software partitioning of a payment terminal

4.3 Memory map

The following assigns the software partitioning to a concrete memory map. As described in Section 4.1, the target hardware uses SRAM as source for data and code execution. This circumstance makes handling of the different sections easier, because code, data and the stack of a context can be grouped together in a continuous memory range.

For this project, an arbitrary split was introduced that divides the memory into secure and non-secure attribution. Each domain is assigned exactly half of the available 256 kB SRAM.

The secure domain houses the library manager, secure interrupts and the isolated libraries. The additionally empty memory of this domain is activated together with the non-secure code and can be used for secure functions of the application. Non-secure memory is, except for the always-on section described in Section 4.3.1, unlimited usable by the application.

Figure 4.2 illustrates the different types of memory sections:

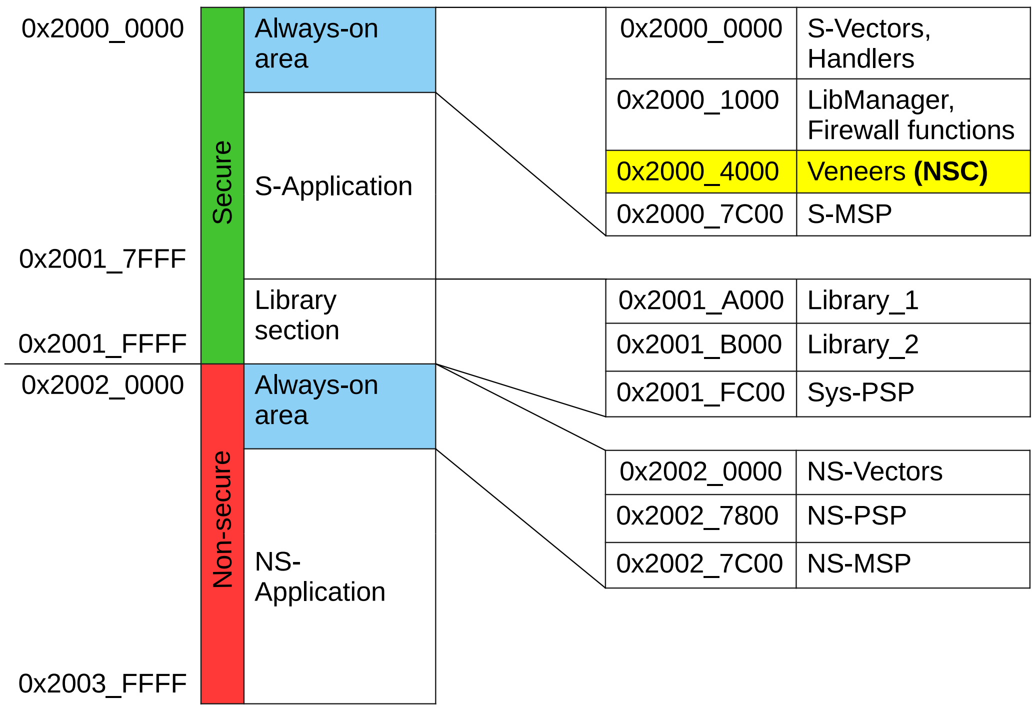

Figure 4.2: Memory map of the target system

4.3.1 Always-on areas

Both security domains have a section, which must never be disabled by the context manager. They contain essential code and data that is needed before the context manager is triggered:

- **Interrupt vectors:** The address of the target interrupt handler is fetched automatically by hardware from this vector. If this memory is disabled by the firewall, an error state is entered by the processor and execution halts.

  Because both security domains have an independent vector table, an always-on region must also be placed in non-secure memory.

- **Interrupt handlers:** Context switching is triggered by a *Firewall-Interrupt*, whose handler implementation must always be enabled. Additionally, all other higher prioritized interrupt handlers must not be disabled by the firewall.

  Even when the *Firewall-Interrupt* is configured by software with the highest priority, some faults are always privileged: *Reset*, *HardFault* and *Non-maskable Interrupt (NMI)* have a hardware-defined priority which is always the highest.

  Handlers for lower privileged exceptions can be deactivated by the firewall: The first instruction within the handler triggers a *Firewall-Interrupt* as nested exception. The context manager performs a switch before resuming the exception. On exception return, another context switch is performed.

- **Context manager entry/exit:** The entry code for the context manager needs to enable the code and data which is required for the complete functionality. Additionally, all security critical sections have to be disabled reliably before returning from the context manager. This small code portion will be added directly into the *Firewall-Interrupt* handler.
- **Secure veneers:** To support the transition from non-secure to secure execution, secure veneers in NSC memory are used. Section 2.2.3 describes this process in more detail. If this section is disabled, the delay of a non-secure→secure call would double, because two context switches would potentially be needed. As the security of this domain transition is already provided by the *TrustZone* mechanism anyway, the context manager is only triggered if the target library is currently disabled.

  This section does also only contain calls of the SG instruction and branches to the secure function address. If it is configured as execute-only memory, malicious code cannot change the function addresses and no information about the real addresses is leaked.
- **Main stack pointers:** The main stacks are used automatically when entering an exception handler. Nested exceptions also store the exception frame on the MSP. When performing a secure→non-secure transition, the secure return address and the IPSR are pushed to the secure main stack (see section 2.2.3). The previously mentioned points highlight the infeasibility of placing the main stacks in disabled sections.

4.3.2 Library section

The current configuration reserves a block of 32 kB for library usage. With the masking capability of the firewall, this section can be divided to house 8 isolated libraries with 4 kB of memory. It is an arbitrary amount for demonstration purposes. More advanced implementations could use individual sections with different resolutions.

Because we are executing from SRAM, the code, data and secure PSP of the context are concentrated into this memory. One slice of this section is reserved for the secure PSP of the application code.

The individual slices are activated exclusively: There is either one library, or the application with the secure PSP activated. Accessing deactivated memory triggers the context manager. Figure 4.3 illustrates the activated memory sections for different scenarios, where the enabled contexts are highlighted in blue color.

4.4 Context manager

The context manager is the central logic responsible for dynamically changing the memory permissions. When an access (data modification or instruction fetch) is performed on a locked memory address, the resulting *Firewall-Interrupt* triggers the sequence of the context manager illustrated in Figure 4.4:

1. Store stack pointer of active context: Depending on the active execution mode and

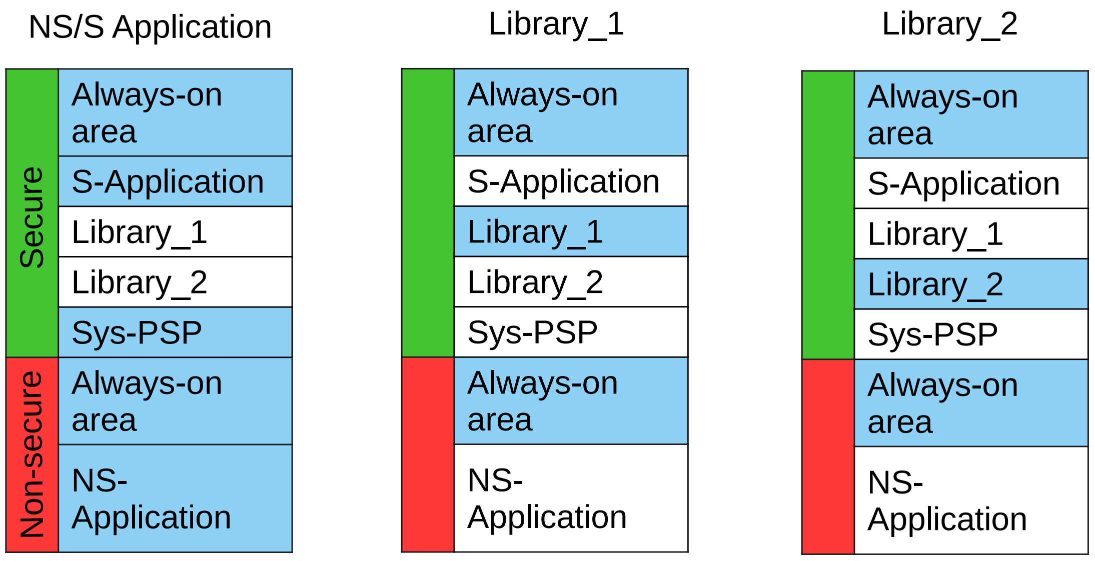

Figure 4.3: Memory activation based on executing code

security domain, one of the four distinct stack pointers was used for the exception frame.

2. Activate code and data sections of the context manager: The memory of the context manager is disabled during execution of (untrusted) code to prevent modifications.
3. Resolve entry reason: Based on the status code contained in the LR, the exact reason of the fault can be replicated.
4. Check access permission, trigger a *HardFault* or other error handling function for unauthorized entry.
5. Store status data and stack pointer to the Task Control Block (TCB) of the source context
6. Clean-up stacks: The exception frame is possibly located on the stack of the old context. It has to be moved to the stack of the destination context to enable the exception-return mechanism.
7. Restore data and stacks of the destination context.
8. Deactivate code and data sections of the library manager to prevent unauthorized modifications during code execution.

4.5 Context transitions

Entry to the context manager is not established through a centralized function or supervisor call. The resulting *Firewall-Interrupt* of an invalid memory access is used for context

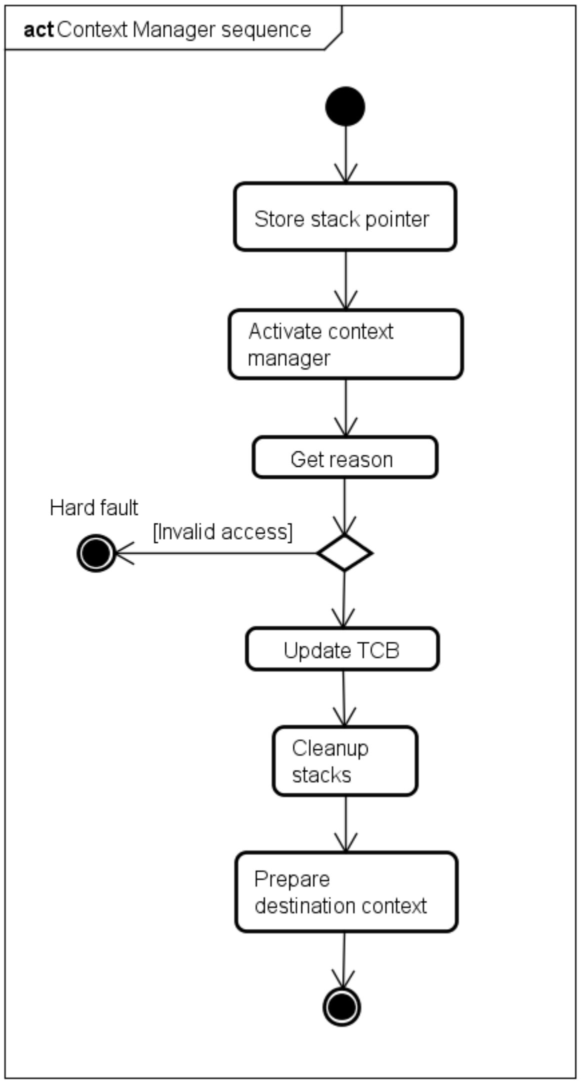

Figure 4.4: High-level sequence of the context manager

switching instead.

Because of the separated security domains and banked stack registers, multiple entry sources to the context manager are possible. This section describes all context transitions which are supported in the basic form. More advanced scenarios, with nested transitions, are demonstrated in Section 5.4 in the *Implementation* chapter.

4.5.1 Non-secure to secure call

One of the most essential transitions is the possibility to call a secure function from the non-secure application. This case deals with application code, running unprivileged with the PSP enabled. The context manager is not concerned if the secure function belongs to the domain of the application as described in Section 4.3.

If the library is currently deactivated, the context manager is needed in addition to the security transition performed by the *Security Extension*.

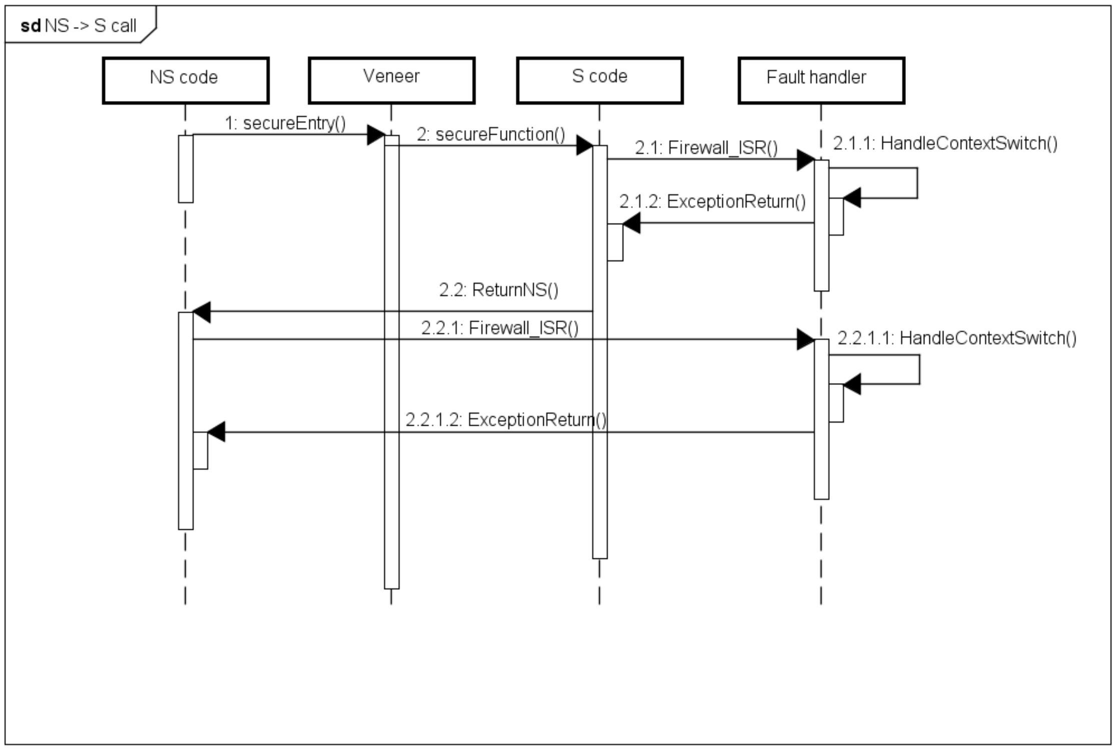

Figure 4.5: Non-secure to secure call sequence

Figure 4.5 illustrates the execution sequence. The secure function is not called directly. A secure veneer, responsible to call the SG instruction, is used instead. Those veneers are located in NSC memory as described in Section 4.3.1.

The branch from the secure veneer targets a memory address, which is disabled by the firewall. The resulting *Firewall-Interrupt* originates from the secure domain with the first instruction of the function as return address. The exception frame is located on the secure PSP of the wrong context. Because this stack pointer is switched to the new context, the stack frame has to be moved to the destination stack. The exception-return mechanism would fail otherwise.

After the context switch and exception-return are performed, the secure function is finally executed. Assumed that the function is not interrupted or calls another context, it can finish without noticing side effects. A return to the non-secure domain is performed with the last instruction.

The memory of the return address was disabled by the firewall, leading to another *Firewall-Interrupt*. This time, the transition to the non-secure execution mode was performed before. The interrupt source is the non-secure domain with the exception frame residing on the NS-PSP. As this stack pointer is not changed by the context manager, the exception frame can remain. Execution resumes at the previously fetched instruction after the exception return.

4.5.2 Non-secure call-back

Another essential transition is calling a non-secure function from the secure domain. This is needed when triggering pre-registered hook functions from secure interrupts or using a utility function provided by the non-secure application.

From the design perspective, this sequence is basically a reverse version of the previous case: The secure code branches to a non-secure address and triggers a change in execution mode. Fetching the instruction causes a *Firewall-Interrupt* which is starting the context manager. As the source of the exception was the non-secure domain, the exception vector is already on the NS-PSP and must not be moved. The additionally pushed return address and control data has to be moved to the secure stack of the application context. More details about this mechanism are given in Section 2.2.3.

When the non-secure code returns, the secure instruction address is automatically fetched from the secure stack and the execution mode changes. Immediately, a *Firewall-Interrupt* is triggered. The exception frame is on a secure location and has to be moved to the target stack. Execution resumes at the secure function after the exception-return.

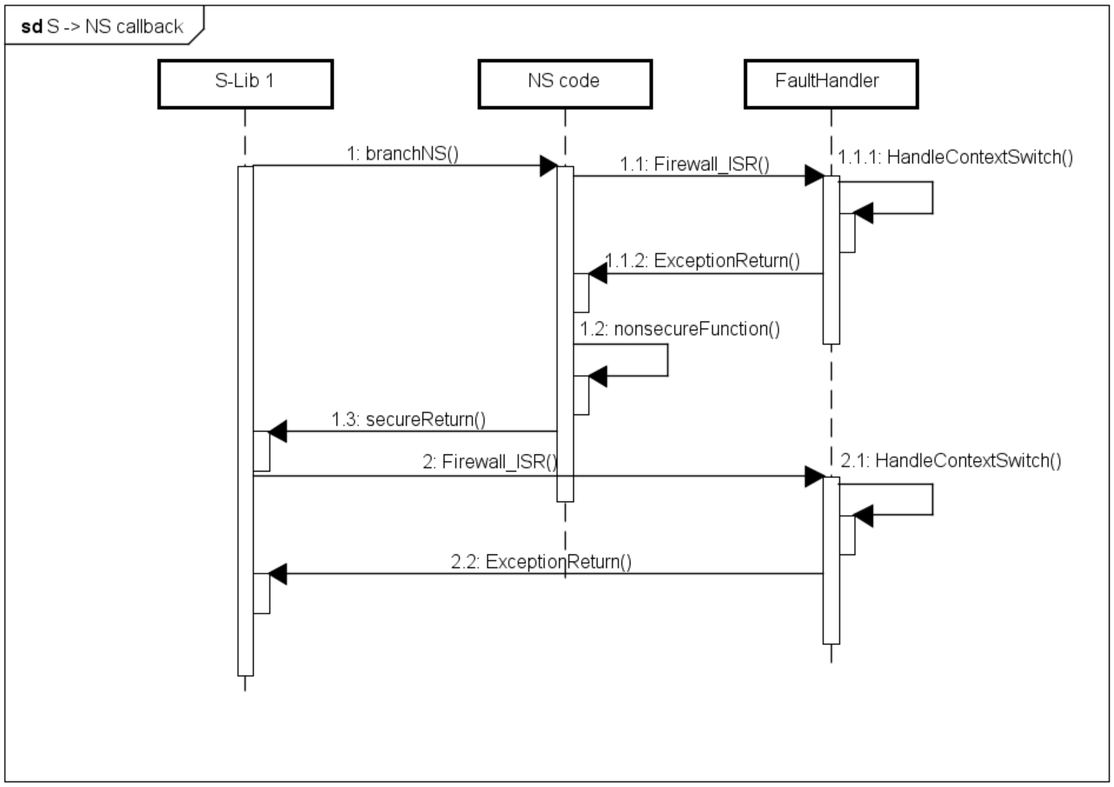

Figure 4.6: Non-secure call-back sequence

4.5.3 Secure → secure call

Calling a secure function of a different context is the first scenario, where only the context manager is concerned. The security extension is not involved, because accesses in the same security domain are completely unrestricted.

This function call is realized with a default branch instruction. When the first instruction of the target is fetched, the *Firewall-Interrupt* is triggered. Switching to a different context in the secure domain requires moving of the exception frame to the stack of the target.

Returning to the caller function behaves similar to the initial call. In essence, only the stack pointer is changed, and the exception frame is moved. The implementation has to distinguish both cases only for access control described later.

In the simplest case, illustrated in Figure 4.7, the called function is finished and returns to the caller. Additional interrupts and nested cross-domain calls are not discussed at this moment. Those scenarios are supported by the design and will be discussed in Section 5.4.

Checking the permission of the function call is the responsibility of the context manager. Section 4.6 discusses this topic in more detail.

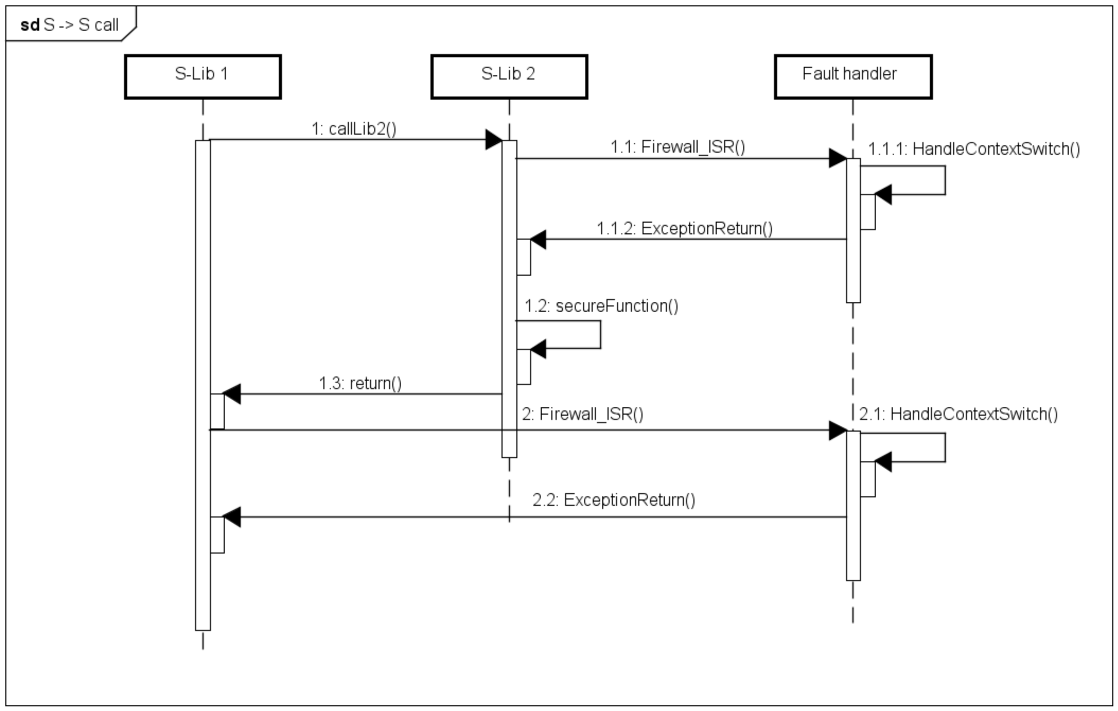

Figure 4.7: S → S call

4.5.4 Exception-based transitions

Exception handlers also belong to a specific context, which might be disabled when the interrupt occurs. This design tries to avoid special restrictions on allowed interrupts. Moreover, blocking or dynamic disabling of interrupts to prevent advanced scenarios must not be used as a workaround. The traditional priority, chaining and nesting features of exceptions also have to be fully supported. All use cases require that the *Firewall-Interrupt* has the highest configurable priority. Context switches required by higher prioritized interrupts would fail otherwise. The following sections summarize the possible exception sequences:

Secure exception during non-secure execution

The first exception use case is a combination of changing all execution modes at once. Non-secure application code, executing as unprivileged task, is interrupted by a secure exception of a different context. In this case the privilege level and security state are changed by the build-in functionality of the core, and context switching is handled by the context manager.

Entry to the secure exception is done in the conventional way with the additional switch of the execution permission. The first instruction of the interrupt handler causes the *Firewall-Interrupt* as nested exception. The exception frame is located on the secure MSP, because execution already switched to privileged mode beforehand. This stack is shared by all tasks anyway, preventing the need of moving the fault stack.

When the initial secure exception finishes, the exception frame is retrieved from the non-secure stack. Fetching the next instruction causes a *Firewall-Interrupt* again. This case behaves the same as returning from a secure function call. No explicit stack modifications are performed, only internal data structures are updated.

Secure exception during secure execution

Figure 4.9 illustrates the scenario, where the execution of secure code is interrupted by a secure exception of an inactive context. The initial exception causes a switch to privileged mode, where the first instruction of the handler triggers a *Firewall-Interrupt* as nested exception. The exception frame of the outer exception is located on the stack of the wrong context and has to be moved. The inner exception happened already in privileged handler mode and placed the frame on the MSP.

Returning from the outer exception causes the same context switch as a return from a normal secure function: The next instruction of the interrupted context triggers the context switch. The created exception frame has to be moved to the stack of the initial context again, because the execution privilege was decreased before the *Firewall-Interrupt* happened.

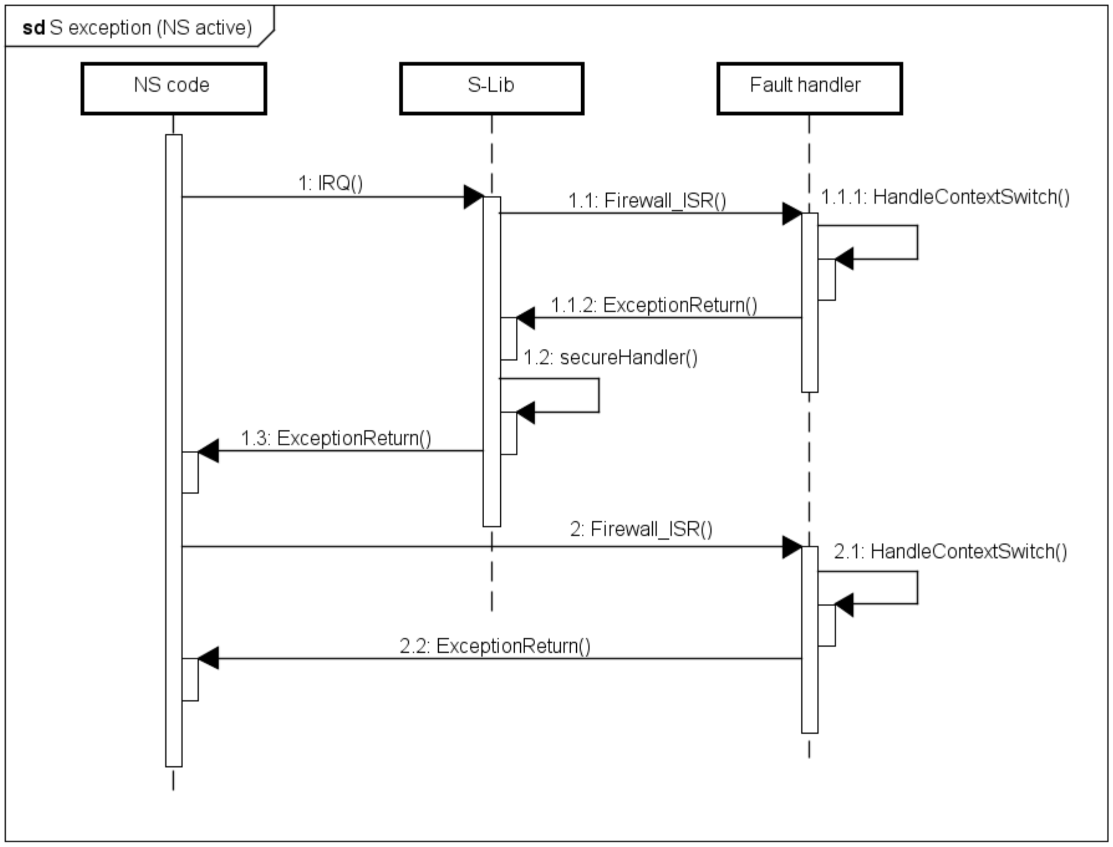

Figure 4.8: Secure exception during non-secure execution

Secure exception nesting

A fundamental concept of exceptions on *Cortex-M* processors, is the nesting of higher prioritized exceptions. A simple design could deactivate subsequent interrupts at the context switch. In contrast, this project tries to preserve the established programmers' model of the original architecture.

The exception priority can be configured freely in software. The only constraint is, that the *Firewall-Interrupt* must have the highest configurable priority. All other exceptions can be nested freely, even if they belong to different contexts. It is assumed that secure interrupts have a globally higher priority compared to non-secure interrupts. This precondition simplifies the stack handling, because no subsequent transitions in the security state are possible. The execution flow of nesting one exception level is illustrated in Figure 4.10. The outer exception is handled as described in previous sections. Whether the exception frame has to be moved depends on the security state of the interrupted function. The context switch of the inner exception does not require changes to the fault stacks. Execution already was in privileged handler mode, resulting in placement of the exception frame on the secure MSP, which is shared by all contexts.

This condition holds true for all subsequent nested exceptions, as the MSP is used exclusively until the lowermost exception returns to the normal execution flow.

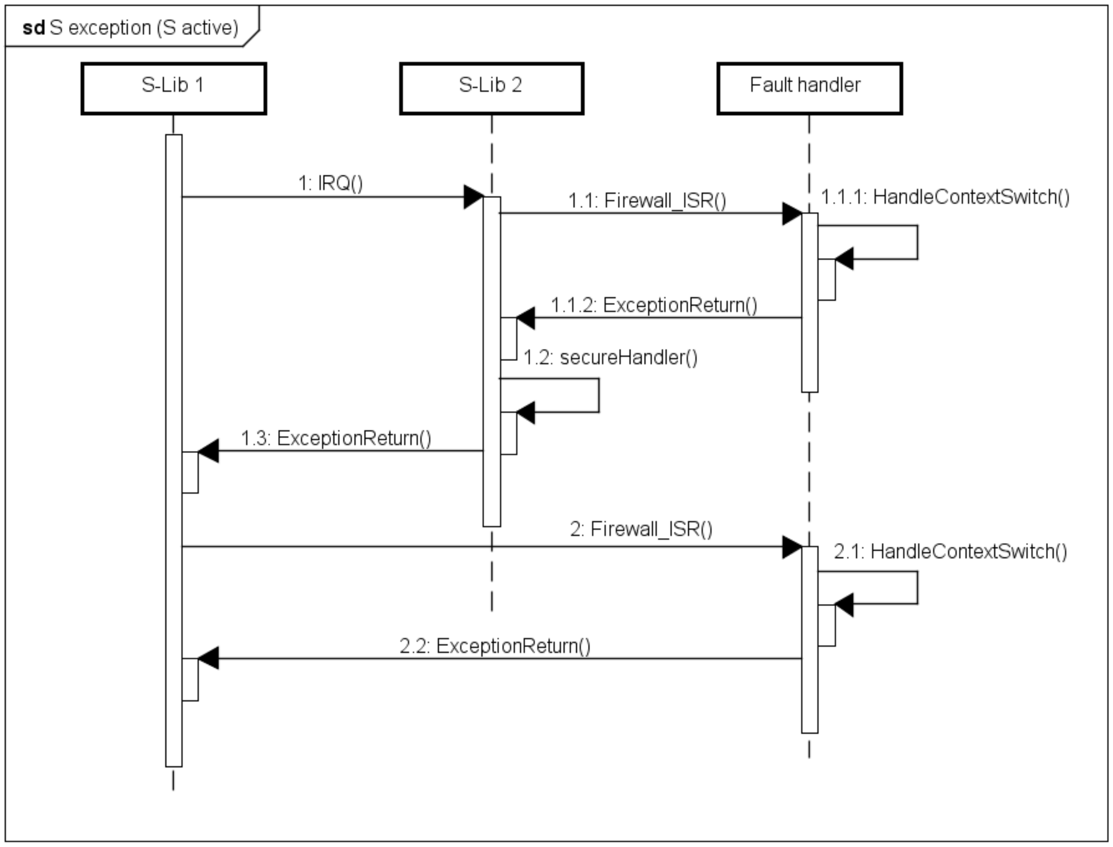

Figure 4.9: Secure exception during secure execution

Function call from exception handlers

All functionality required for processing an interrupt might not be implemented in the handler. An additional function from a secure library might provide utilities or can be used to pass data for later usage.

When calling the function, which is located in another context, a *Firewall-Interrupt* is triggered which handles the task switch. The entry behaves the same as for other nested exceptions, with the exception frame located on the MSP. No stack modifications have to be performed, because the execution mode remains privileged. When the function returns eventually, the task switch is the same scenario in reverse.

This use case is mostly mentioned for the sake of completeness, because a lot of care must be taken. One concern deals with performance considerations: Exception handlers should be as short, and therefore as fast as possible. Calling a utility function, which might be extended for an unrelated reason at a later time, could unintentionally influence the real-time properties in the future.

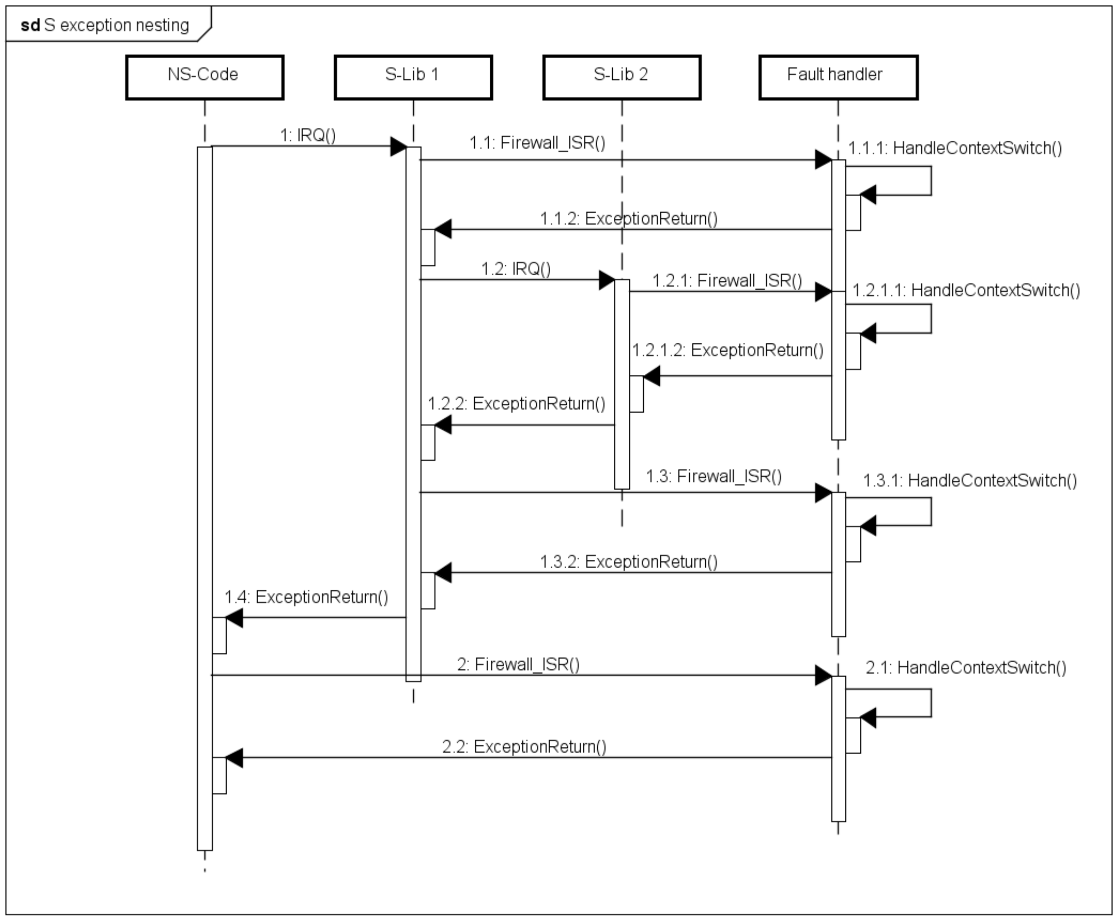

Figure 4.10: Secure exception nesting

Another serious issue arises from the execution privilege. The exception handler is executed as privileged function. With this status, it has full access to all system registers, configurations and the MSP. When an external function is directly called by the handler, the privilege is carried over. Returning to the calling exception handler is the responsibility of the function. The context manager cannot check the correctness of the return address. For this reason, calling an untrusted function from an external library, creates a serious security weakness.

A proposed solution to prevent both issues could be realised by using the Pended Service Call (PendSV) or a periodic task, which performs the complex calculations. The interrupt handler performs only the minimum required functionality, e.g. reading data from external peripherals and storing them in buffers, and registers a PendSV or sets an implementation specific flag. Data processing is only activated once all interrupts are completed. The PendSV additionally has to decrease the privilege level and modify the return stack, in order to return to the initially interrupted function.

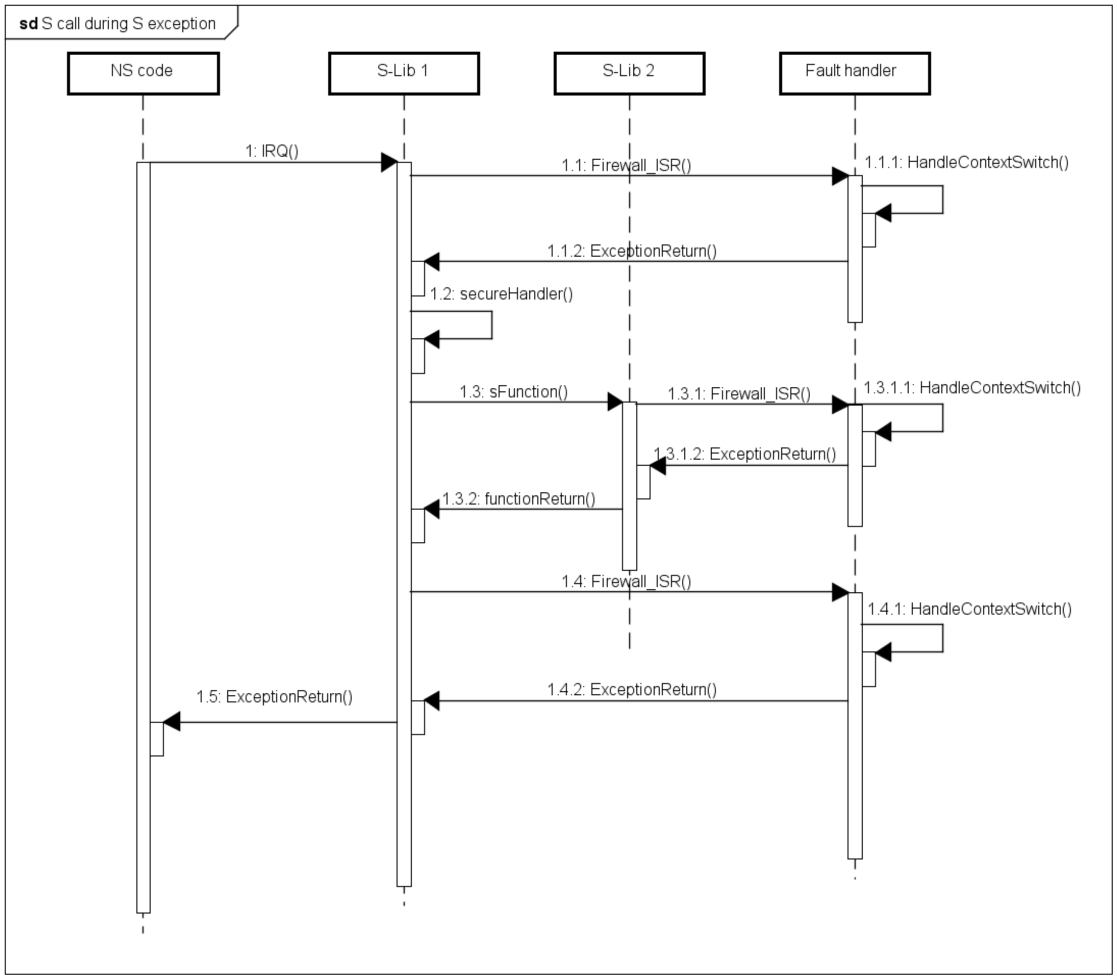

Figure 4.11: Secure call from secure exception handler

Non-secure exception during secure execution

Normal thread execution in the secure domain does not influence the occurrence of non-secure interrupts. They can happen all the same and perform a domain transition as described in Section 2.2.3.

The extended exception frame of the initial exception is placed on the currently active secure PSP. It has to be moved to the stack associated with the application context, to enable the exception-return sequence. The exception frame of the *Firewall-Interrupt* is located on the non-secure MSP, because switching the security domain and execution privilege was already finished beforehand. Returning from the non-secure handler uses the data of the extended exception frame. The required task switch behaves the same as returning from a non-secure call-back.

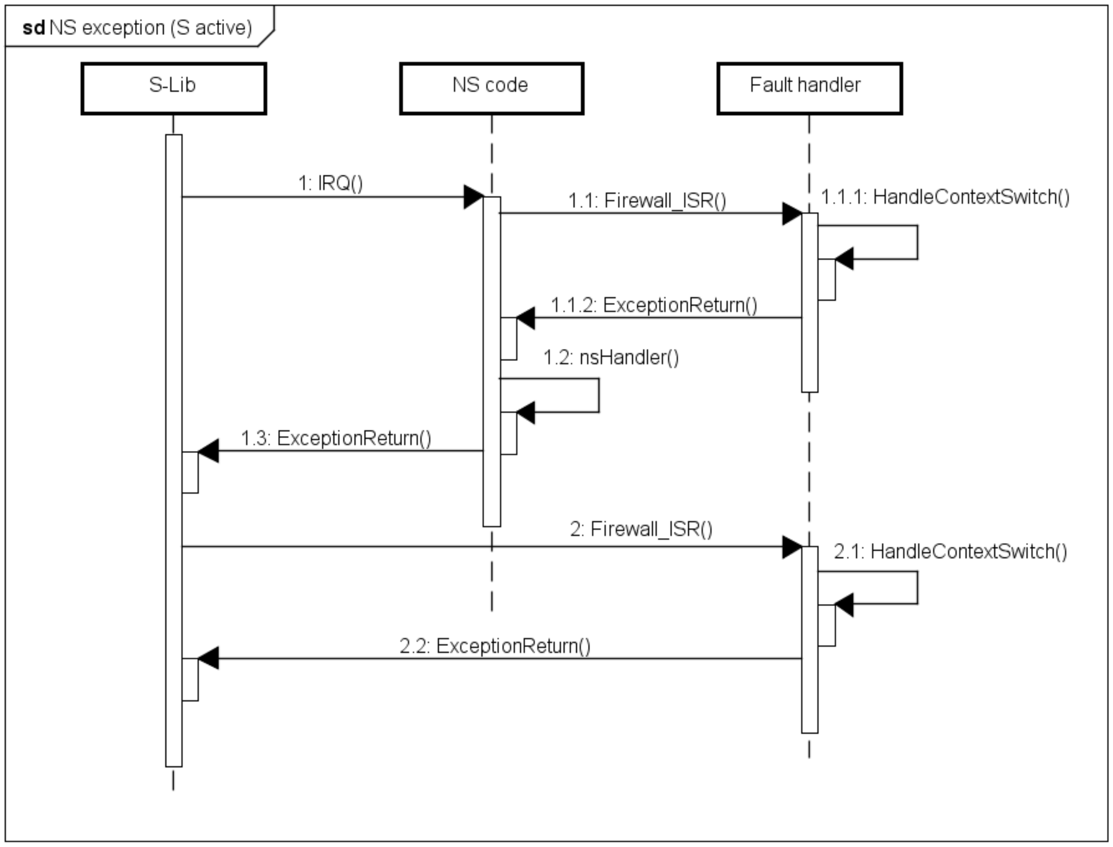

Figure 4.12: Non-secure exception during secure execution

4.6 Sanity check

The strict isolation between the two security domains is already handled by the default *Security Extension* of the processor. Information leakage and invalid accesses from contexts inside the secure domain have to be additionally checked by the context manager. As the implementation is very application dependent, only basic checks are performed within the scope of this project. The following considerations give an overview of the responsibilities imposed on the context manager.

Information about the memory map is the essential requirement of the context manager. Only with a defined schema, the matching of fault-addresses and context borders can be performed. The simple static design, which is described in Section 4.3, was chosen to simplify the implementation and reduce the effort of debugging. More advanced projects could use linker symbols to retrieve this information automatically, and setup variably sized sections at the initialization stage.

Only function calls are allowed between secure contexts. Read/write operations on disabled memory regions are prohibited by this design, passing of large data has to be performed with shared buffers in always-on regions described in Section 4.3.1. Noticing such an invalid access is simple: The resulting *Firewall-Interrupt* contains a *PC* value belonging to the already activated context. This means, that the executed instruction was in an activated section, but another side-effect (e.g. data access) triggered the fault.

A more advanced logic is needed to guard the function calls. Not all functions of a context should be publicly available to external calls and branching to the middle of the function should be prevented. This could be exploited to skip the data validation part of a function and perform actions on an invalid input. The resulting side effects could further cause vulnerabilities in the security architecture. The first step of securing the entry points can be realized with the help of function whitelists: Every context registers the publicly available functions at the context manager. Those lists are stored internally and are queried when context switches are performed. A proposed design would place allowed function pointers at the beginning of the memory section with the help of compile-time macros.

When a function of a disabled context is called, the resulting *Firewall-Interrupt* indicates the start address of the function. Preventing context switches on addresses, which are not on the whitelist of the destination context, would solve both requirements of the sanity check. Unfortunately, the flexibility of secure code would be limited severely by this simple implementation:

- Returning to the source context from a function call also triggers the context manager. The source address is the instruction following the initial function call, which would fail with the pure whitelist-approach.

  As a workaround, the previous instruction could be inspected. Containing the signature of one of the branching operation codes could indicate a return in the execution flow. This approach would loosen the requirements by also allowing arbitrary calls to all instructions of this pattern.

- Code execution can always be interrupted by an exception. Context switches, caused by exceptions, have to be traced to enable the default exception-return procedure.

The previous mentioned points highlight the need of detailed information about the context switch reason, and the management of a call-stack. The context switch reason is mainly derived from the fields of the *EXCEPTION_RETURN* value located in the link register. The call-stack collects the initial return addresses before a context switch was performed. If the source address of the *Firewall-Interrupt* equals the top of the call stack, return from a nested call is identified. Sections 5.3.2 and 5.3.1 give the implementation details of those components.

Chapter 5

Implementation

This chapter documents the details of the practical implementation and the used toolchain during project development. The general software architecture and the verification steps of the proposed solution are also covered.

5.1 Toolchain

The main software development was done with the *Keil μVision* Integrated development environment (IDE) and the associated compiler, officially developed by *ARM*.

The decision, to use these commercial products instead of free and open-source solutions (e.g. GNU Compiler Collection (GCC)), was made because of the following reasons: *ARM* primarily releases examples and sample projects with this toolchain. As the target architecture is not yet used in the mainstream market, support and documentation from other sources is still very limited. The initial orientation and project launch was made easier, because the well-structured information from a single resource could be used.

Another crucial factor was the company internal support from *NXP*. The same toolchain is used for other projects targeting this hardware. The licence, support packages and a sample project was provided as initial starting point. The re-use of the implementation results is also simplified if the same workflow is retained.

Creating a new project for an *ARMv8-M* target, two distinct projects are created and the required support files for the specific hardware are linked automatically. The reason for two projects is the better separation of secure and non-secure code. Unintentionally calling of non-secure code with increased privileges is made harder and secure code is not compiled into the non-secure binary.

The secure project is compiled/linked first and produces two objects:

1. The stand-alone binary contains all the code and constant data of the project. Initialization and a bootloader should also be included in this file. If the application does not use an additional non-secure component, the complete system is contained in this binary.

2. An import library is produced, to allow linking of the non-secure code. It contains the symbol table for all the secure functions which can be called from non-secure

code. They point to the static addresses of the entry functions located in the NSC memory section.

This sequence has a direct impact to the software architecture: Call-backs to the non-secure domain are realized with function pointers. They are passed on a per-call basis or statically registered with a start-up sequence.

Functions decorated with the *non-secure-entry* attribute cannot be optimized out, because there is no information about the subsequently compiled non-secure application. This could add a lot of unused code unintentionally, when including components from external libraries.

When the secure code is updated at a later stage, care has to be taken to avoid incompatibilities with the already deployed non-secure application. New secure functions can be added without causing recompilation of non-secure code which does not use this newly implemented feature. The old import library has to be passed to the linker, to prevent reordering of the secure veneers. New entry functions are appended at the end. Assuming that the address of the NSC section and the memory map remained the same, only the secure project has to be rebuild and deployed.

5.2 Project architecture

An example project was used to implement and test the context manager. Figure 5.1 illustrates the structure.

The secure code contains initialization, firewall configuration, secure interrupts and context manager. Additional functions were implemented as secure libraries to test the non-secure entry and callback scenarios. In contrast to a real-world application, all secure code was developed in a single project and compiled at once. Based on the memory map defined by the scatter file, context and IP protection is only ensured during execution.

This approach was chosen to simplify development and debug operations of the context manager. It also allowed quick deployment of the binaries without the need of a bootloader.

The non-secure code implements several use cases to verify the implementation. More details about the verification are given in Section 5.4.

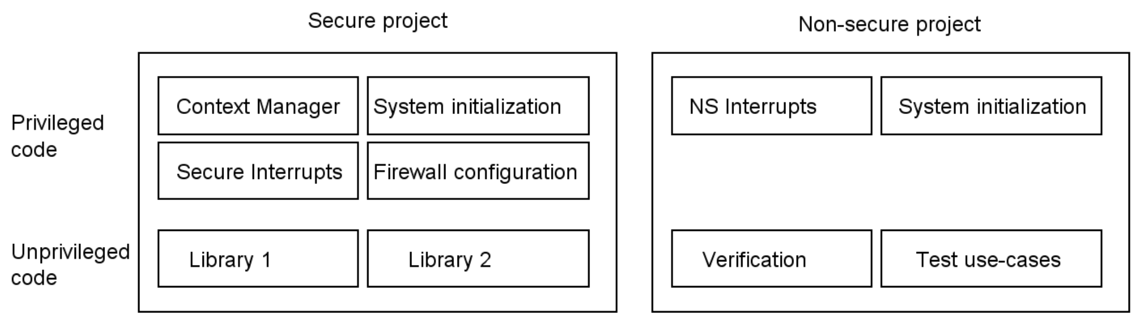

Figure 5.1: Project structuring

5.3 Context manager

The context manager is the central part of this project. The design described previously was implemented as proof-of-concept. Details of the concrete implementation of the essential parts are given in Section 5.3.3.

The main functionality of the context manager is implemented in four steps:

- Identifying the cause of the context switch is the essential operation. The *EXCEPTION_RETURN* value and registers from the exception frame are used for this decision.
- The call-stack keeps track of context switches caused by nested calls. Returning from an inner function is detected to perform stack clean-up operations and allow otherwise prohibited task switches.
- When switching the stack pointer, the exception frame also has to be moved. Based on the fault reason, the location and size of the exception frame is different.
- Finally, the firewall configuration has to be changed for allowing the execution of the destination context.

5.3.1 Fault reason

The exact fault reason is required for the sanity check and all later stack management. The main indication of the exception source is given with the fields of the *EXCEPTION_RETURN* value. It is written to the link register on exception entry. A detailed description about the available fields is given in Table 2.3.

The values of the link register and program counter placed on the exception frame are also needed for this identification. The program counter points to the execution address, which triggered the *Firewall-Interrupt*. With the link register, nested exceptions can be identified. Algorithm 1 gives an implementation overview.

5.3.2 Call-stack management

The call-stack is stored to get the information about returning from a function or interrupt handler. The general-purpose registers are restored when a return is detected.

Updating of the stack and the location of the return address is dependent on the fault reason. The used fault stack location can be derived from the *EXC_RETURN* value. The relevant fields are *S* (secure/non-secure, bit[6]) and *SPSEL* (stack pointer selection, bit[2]). With this information, the return address can be pushed to the call-stack. Algorithm 2 gives an overview of the required steps.

5.3.3 Detailed transition sequences

This section combines the previously mentioned tasks and gives a detailed summary of all required steps to handle a context switch.

Algorithm 1 Algorithm to get fault reason.

```
1: procedure GetFaultReason($exceptionReturn$, $faultstack$)
2:     $linkRegister := faultstack[5]$
3:     $programCounter := faultstack[6]$
4:     $faultReason := NULL$
5:     if $linkRegister == 0xFEFFFFFF$ && !$SecureOrigin$ then
6:         $faultReason := REASON\_NS\_CALLBACK$
7:     else if $CallstackTop == programCounter$ && $SecureOrigin$ then
8:         $faultReason := REASON\_RETURN$
9:     else if $CallstackTop == programCounter$ && !$SecureOrigin$ then
10:        $faultReason := REASON\_RETURN\_NS$
11:    else if !$SecureOrigin$ && $PrivilegedOrigin$ then
12:        $faultReason := REASON\_NS\_IRQ$
13:    else if $PrivilegedOrigin$ && $SecureOrigin$ && $linkRegister == EXCEPTION\_PREFIX$ then
14:        if $linkRegister ==$ !$PrivilegedOrigin$ then
15:            $faultReason := REASON\_S\_IRQ$
16:        else
17:            $faultReason := REASON\_S\_IRQ\_PRIVILEGED$
18:        end if
19:    else if $PrivilegedOrigin$ && $SecureOrigin$ then
20:        $faultReason := REASON\_S\_PRIVILEGED\_CALL$
21:    else
22:        $faultReason := REASON\_CALL$
23:    end if
24:    $returnfaultReason$
25: end procedure
```

Secure function call

Calling a secure function from a non-secure context is one of the most basic use cases. The implementation has to track this case, to verify the context switch when returning to the non-secure domain again.

Figure 5.2 illustrates the detailed steps of this use case.

1. The non-secure code calls the function by the name declared in the provided header file. The import library does not point to the implementation of the function directly - a branch to the secure veneer located in NSC-memory is performed instead. At this function call, the security domain is changed to the secure state. It only succeeds, if the target instruction is the SG operation.

2. The secure veneer calls the concrete implementation of the secure function immediately. By using a direct branch, the LR still points to the next instruction of the non-secure code. No return to the secure veneer will be performed.

Algorithm 2 Call stack management

```
1: procedure UpdateCallStack($exceptionReturn, faultStack, faultReason$)
2:     $linkRegister = faultStack[5]$
3:     if $faultReason$ $IN$ $[REASON\_RETURN, REASON\_RETURN\_NS]$ then
4:         $PopFromCallstack()$
5:     else if $faultReason$ $IN$ $[REASON\_CALL, REASON\_S\_PRIVILEGED\_CALL]$
   then
6:         $PushToCallstack(linkRegister)$
7:     else if $faultReason == REASON\_S\_IRQ$ then
8:         $PushToCallstack(originalFaultStack[ReturnAddress])$
9:     else if $faultReason == REASON\_NS\_IRQ$ then
10:        $PushToCallstack(extendedFaultStack[ReturnAddressExtended])$
11:    else if $faultReason == REASON\_NS\_CALLBACK$ then
12:        $PushToCallstack(secureStack[0])$
13:    end if
14: end procedure
```

3. Because the secure function is located in another context, a *Firewall-Interrupt* is triggered when fetching the first instruction. The *EXC_RETURN* value located in the LR indicates secure, unprivileged code as the exception origin. The exception frame has to be moved before changing the secure PSP to the destination context.

   The cleaned stack pointer of the switched-out context is updated in the associated TCB. To detect the return to the non-secure code later on, the old LR value located on the exception frame is pushed to the internal call stack.

4. The default exception return procedure restores the stacked register values from the PSP of the active context. Execution of the secure function can be performed, because the firewall rules were updated by the context manager beforehand.

5. To return from the secure domain, the *BXNS* instruction is used. Clearing of security critical registers has to be performed manually beforehand. Secure code has full visibility over the non-secure memory. For this reason, the non-secure return address was initially passed in the LR. It was preserved by the function and can be used directly to return from the subroutine call. Fetching the next instruction of the non-secure code will inevitably cause another *Firewall-Interrupt* and activation of the initial context. This *return*-sequence is described in the next section.

Function return

A function return has to be distinguished from other cases for the sanity check described in Section 4.6. In combination with the previous sequence, a secure API call can be handled. This section focuses on the return of a subroutine call, the secure call is only indicated in Figure 5.3.

1. An initial secure call is needed to prepare the inner state of the context manager for a function return. This step includes the switch to the secure domain, branching from

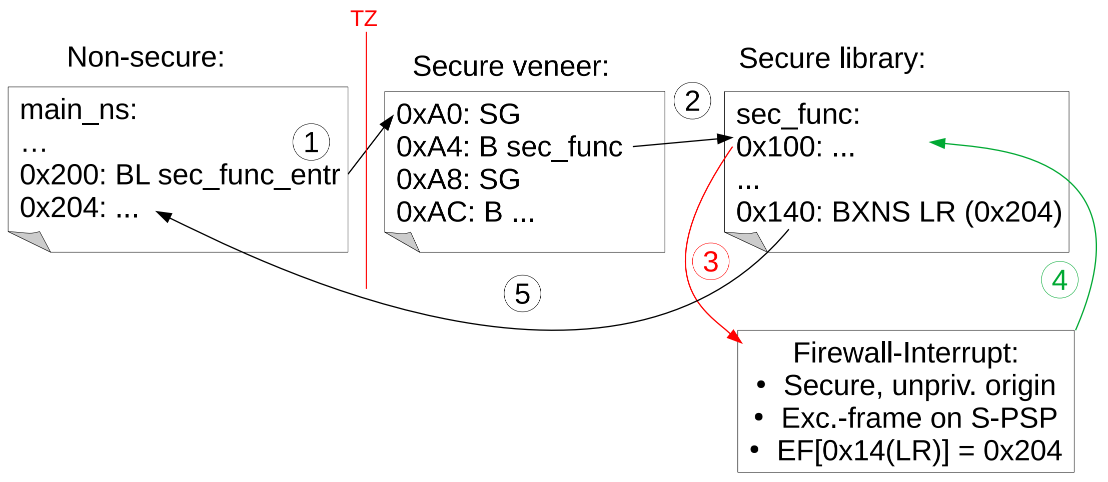

Figure 5.2: Secure call sequence

the secure veneer and switching the context to the target. The *BXNS* instruction starts the return-sequence.

2. Fetching of the next instruction causes a context switch to the initially active non-secure code. The transition to the non-secure domain was finished beforehand, causing the exception frame to be located on the NS-PSP. This stack is not modified by the context manager and requires no moving of the exception frame.

   The return address causing the *Firewall-Interrupt* matches the top of the call-stack, indicating the return from a nested call.

3. The exception-return mechanism automatically cleans the exception frame from the non-secure stack. Code execution continues oblivious to the occurred context switches.

Depending on the security state of the calling code, we have to further distinguish the stack handling. Returning to secure code is handled in following sections.

Secure function call from secure code

Calling a secure function in a different context also initiates a context switch. When the calling function is already in the secure domain, the *TrustZone* features are not involved. Managing the access rights is only ensured by the firewall implementation. Figure 5.4 illustrates this sequence in more detail.

1. Because both contexts already are in the secure domain, the secure function is directly called in this case. The called address was supplied from a function pointer, or linked from an export library. A sanity check has to performed in the context manager to prevent unauthorized context entry.

2. Fetching the first instruction of the other context causes a *Firewall-Interrupt*. There is no difference to the non-secure call visible to the context manager: The exception

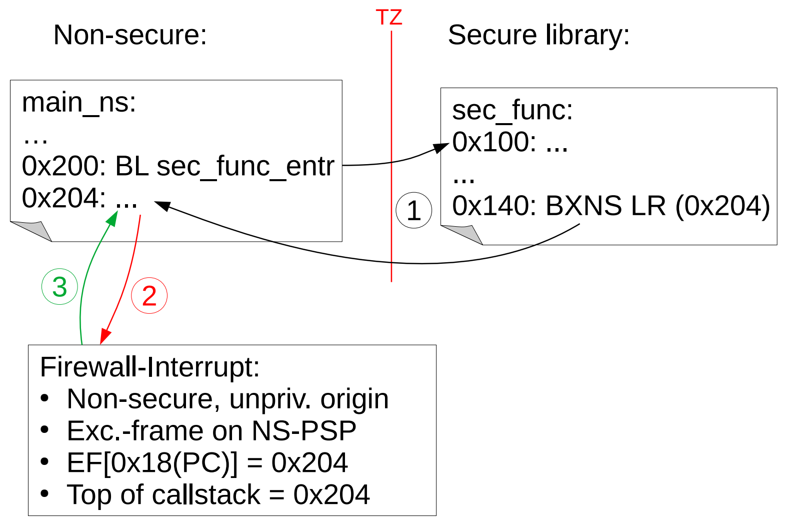

Figure 5.3: Secure return to non-secure code

source is also secure, unprivileged code. The return address from the exception frame is pushed to the call stack. Before switching the PSP to the new context, the exception frame has to be moved.

3. The exception-return procedure uses the restored registers to return to the called function.

4. Returning from the subroutine-call uses the default *branch* instruction. It is assumed, that the function does not tamper with the return address. The context manager cannot distinguish a subroutine call from an invalid return procedure. Validation can only be performed on function entry, because no information about the function length is known.

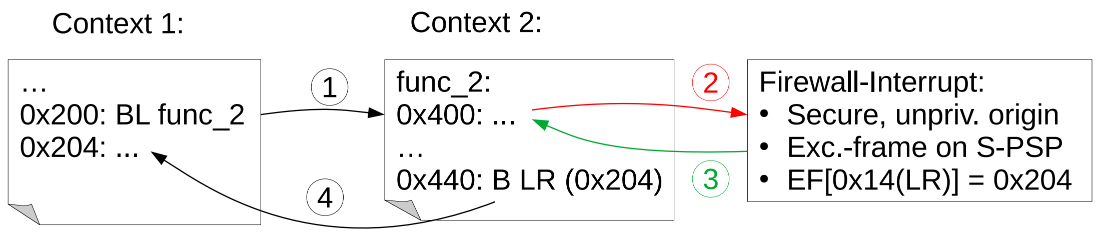

Figure 5.4: Secure call from a secure context

Secure return to secure code

Returning to a secure function from a secure subroutine-call also requires different handling from the context manager. The management of the call-stack is an essential requirement for this case.

1. Calling and returning from the function take place without changing the security status of the processor. The switch to the second context was explained in the previous section and is omitted here.

2. The operation after the initial branch causes a context switch. The *EXC_RETURN* also signals entry from secure, unprivileged code. Looking at the internal call-stack helps to distinguish this case from previously mentioned entry reasons. After moving the stack frame and activating the initial context, the exception-return can be performed.

3. The build-in functionality of the core cleans the stack frame and restores the register values.

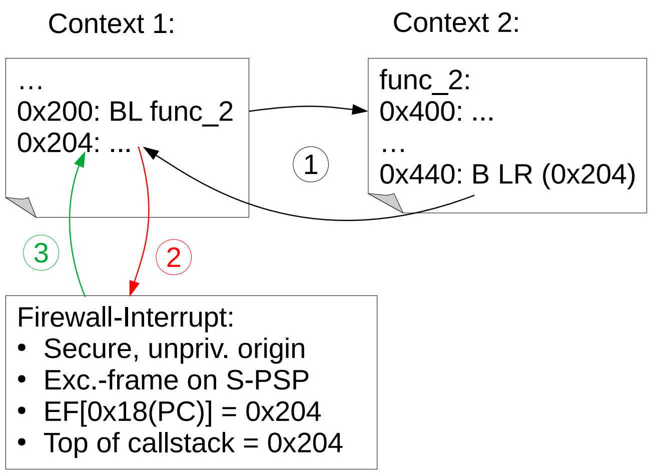

Figure 5.5: Secure return to secure code

Non-secure call-back

The core architecture provides basic support to enable non-secure call-backs without causing security vulnerabilities. Illustrated by Figure 5.6, this advanced scenario has to be specifically handled by the context manager:

1. To trigger the correct internal measures for the switch in execution mode, the *BLXNS* instruction is used. When developing in the *C* language, call-back pointers have to be decorated with the *cmse_nonsecure_call* attribute. This causes creation of the correct instructions and additional code to save security critical registers.

   In addition to the transition of the security domain, the secure return address and status registers are pushed to the secure PSP. A special value, called *FNC_RETURN* is stored in the LR. Section 2.2.3 gives more details about this functionality.

The privilege level of the executing non-secure code depends on the value of the banked *CONTROL* register. In this project, we always assume unprivileged execution of application code.

2. The triggered context switch has a non-secure, unprivileged origin. The exception frame is located on the non-secure stack and has not to be moved for this reason. However, the secure return address and status register has to be moved to the secure destination stack, to enable the return sequence later on. This return address also has to be pushed to the call stack.

3. The exception return restores the stacked register values again. The *FNC_ RETURN* magic number is placed in the LR again to indicate the special return sequence.

4. Branching to the *FNC_ RETURN* value automatically pops the actual secure return address from the stack. Additionally, all banked registers are switched together with the security status. All other registers, which have been cleared for security reasons before the call, have to be restored again manually.

5. The context switch required to continue operation follows the same sequence as the previously mentioned *secure-return* sequence.

   One distinction remains: Because the original registers of this function were not yet restored, the stacked LR on the exception frame still has the *FNC_ RETURN* value. It must not be used as a primary identifier for the fault reason. Nevertheless, this anomaly is discovered, because a non-secure call-back targeting secure code is not possible.

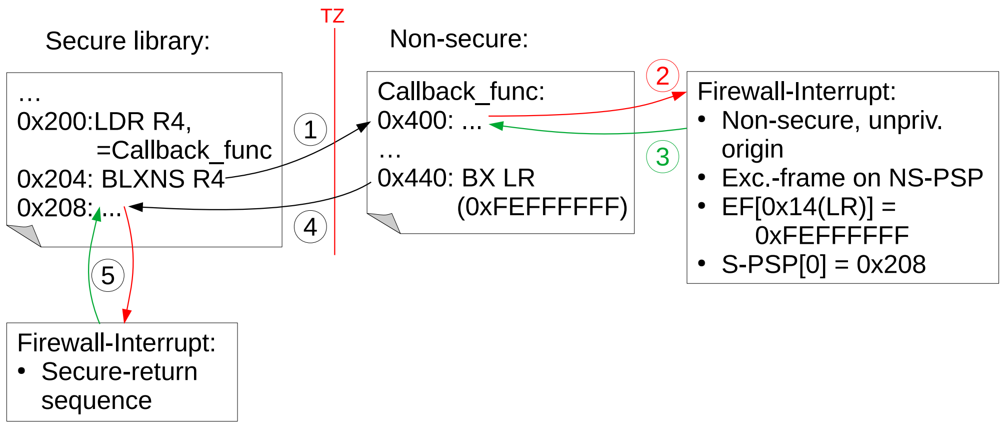

Figure 5.6: Non-secure call-back sequence

Non-secure exception

Non-secure exceptions can happen unpredictable at all times. If the application context is already active by chance, the context manager is not involved and only the default exception sequence is completed. Execution of secure code can also be interrupted unexpectedly.

Multiple features of the build-in architecture ensure the security constraints of the system in this case.

Secure exceptions are considered to be prioritized over non-secure exceptions by the current design. For this reason, interruption of secure, privileged code is not considered as a valid sequence.

Handling of non-secure exceptions is illustrated in Figure 5.7:

1. Secure code has no higher priority than non-secure application code. This allows even the lowest prioritized exception to interrupt secure execution. When crossing the security domain, additional steps are performed internally: In addition to switching the banked registers, an extended exception frame is created on the secure stack. It additionally contains the secure values of the general-purpose registers. Table 2.5 lists all fields in more detail.

2. Before the interrupt handler can be executed, the non-secure context has to be enabled. The context switch is triggered by a nested exception, which again crosses to the secure execution domain. Because the switch of the execution mode was already concluded before the instruction fetch, it is entered from a non-secure, privileged origin. The inner exception frame does not need a relocation, because it is on the non-secure MSP. On the contrary, the extended exception frame of the original exception has to be moved to the secure PSP of the (NS)-application context. The actual Program Counter (PC) of the interrupted secure code has to be pushed to the call stack to detect the exception return later on.

3. The exception return procedure uses the exception frame from the non-secure MSP. The extended exception frame on the secure stack is not touched for this sequence. The restored value of the LR contains the required *EXCEPTION_ RETURN* value to trigger the change in execution mode at return.

4. Returning to the secure domain is done automatically when the *EXCEPTION_ RETURN* value is decoded. The extended exception frame is used to restore system registers (e.g. program counter) and the additionally stacked general-purpose registers.

5. Another context switch is required to continue the initially running code. This requires the same procedure as returning from a normal function call.

Secure exception during secure execution

Secure interrupt handlers can also be located in a deactivated context. The respective context has to be switched-in before the exception can be handled.

1. The exception frame of the initial interrupt is placed on the PSP of the active context. Because execution is already in the secure domain, only a switch in the privilege level is performed.

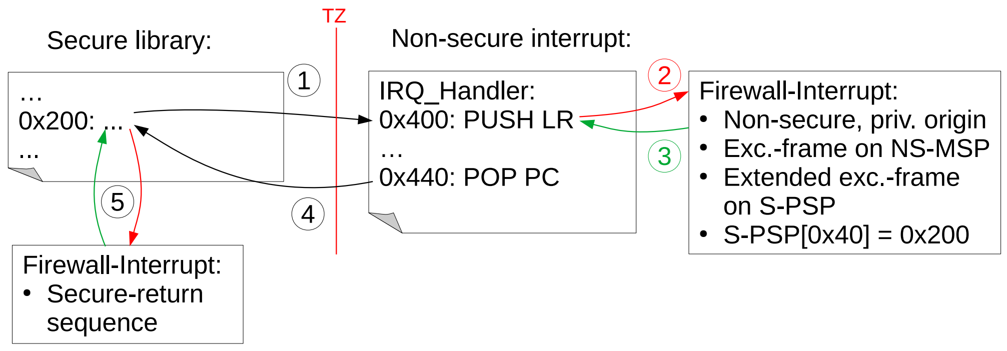

Figure 5.7: Non-secure interrupt sequence

2. The *Firewall-Interrupt* triggering the context switch originates from secure, privileged execution state. The inner exception frame has not to be moved, because it is placed on the secure MSP.

   This case is identified by the stacked value of the LR. It contains an *EXC_RET* value, which requires another exception to be happened previously. The exception frame of the initial exception has to be moved to the PSP of the destination context. It contains the PC of the interrupted code. This address is pushed to the call-stack to identify the exception return later.

3. Returning from the inner exception cleans the exception frame from the main stack. The execution mode stays secure and privileged.

4. Based on the *EXC_RET* value of the outer exception, the registers are restored for continuing the execution of the secure code.

5. Because the return address is located on top of the call-stack, activating the initial context follows the *secure-return* sequence already described.

Secure exception during non-secure execution

A secure exception interrupting non-secure code, is similar to the previously described sequence. Because the change in security mode is handled by the hardware of the base-architecture, no extra overhead is required from the context manager.

Stacking of the exception frame is performed before switching the security state. For this reason, the exception frame of the outer interrupt is located on the non-secure stack. The current execution privilege of the non-secure code influences the whether the PSP or MSP is used.

This changed location has the following impact for processing the sequence:

- None of the two exception frames have to be moved by the first context switch. They are both located on stacks, which are not changed by the context manager.

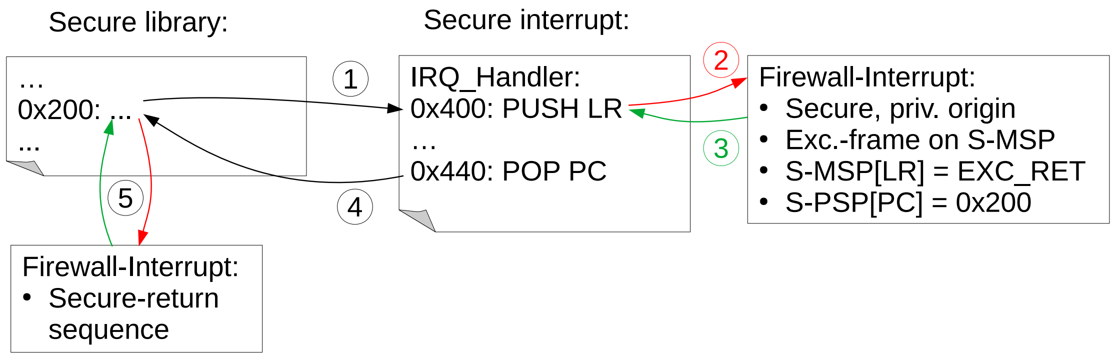

Figure 5.8: Secure interrupt during secure execution

- The return address for call-stack management has to be retrieved from the non-secure stack. For selecting the correct stack, the initial *EXC_ RET* value has to be inspected.
- Returning to non-secure code also changes the execution mode before causing the *Firewall-Interrupt*. The resulting exception frame on the non-secure stack requires no copy operations.

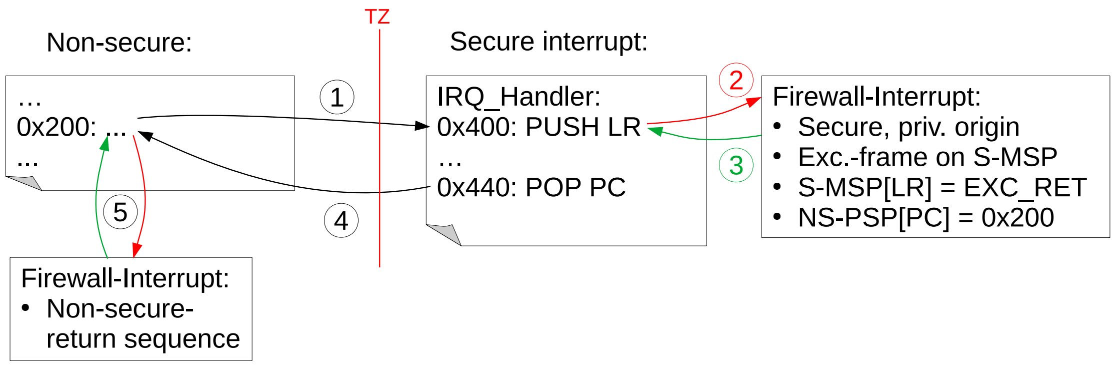

Figure 5.9: Secure interrupt during non-secure execution

Privileged function call

Calling a secure function from a privileged context requires special handling. As described in Section 4.5.4, this functionality should be used with care, to minimize the risk of exploiting the resulting security vulnerability. Although the usage of this function is discouraged, implementation is done for completeness and to avoid a system crash when it is used for legitimate purposes.

1. Calling a function from inside an interrupt handler preserves the privileged execution level.

2. Entry to the context manager, from a privileged and secure origin, wrongly hints to a secure interrupt as the cause. The valid return address in the LR field of the exception vector correctly identifies this nested call. It would contain an *EXC_RET* value otherwise. This return address is also pushed to the call-stack to identify the return to the interrupt handler. Because only the secure MSP is involved, the exception frame is not moved.

3. Returning from the context manager does not change the execution privilege. This code still runs with privileged and secure status. This circumstance illustrates the previously mentioned security vulnerability, because the function can potentially access the complete system.

4. The subroutine is finished with one of the basic return strategies.

5. Resuming the secure interrupt causes the *Firewall-Interrupt* as a nested exception. The interrupt source is again secure and privileged code. The PC field of the exception vector matches the top of the call-stack, which identifies this privileged-return sequence. The exception frame is not moved and will be cleaned by the exception-return.

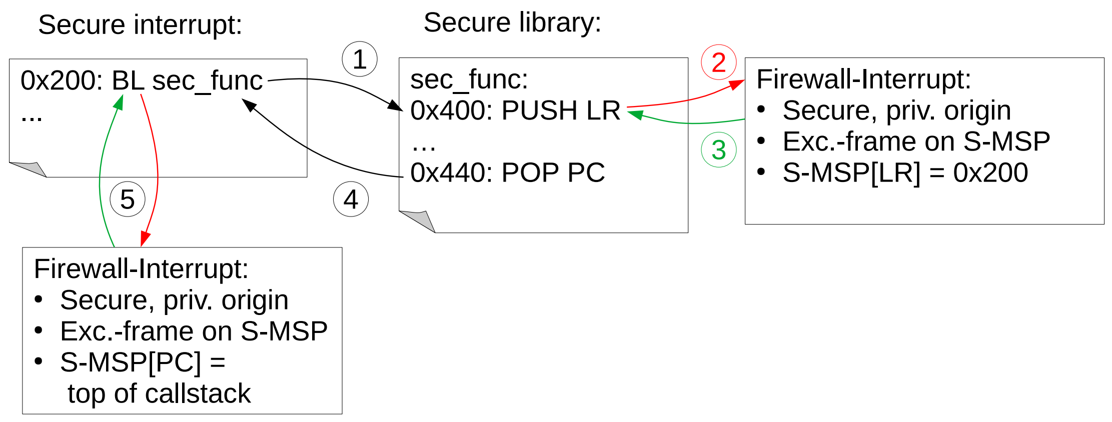

Figure 5.10: Privileged function call

Nested secure exception

Like all non-secure code, exception handlers of the non-secure domain belong to the same context. Nested exceptions do not require a context switch for this reason.

Secure interrupt handlers can potentially be implemented in different contexts. Nesting of interrupts and required context switches have to be supported for this reason.

Figure 5.11 illustrates handling of this sequence in detail:

1. Entry to the first exception level follows the already described sequence. In this example, the exception handler is additionally interrupted by a higher privileged exception. The exception frame used for returning is placed on the secure MSP in this case.

2. Assumed that the second exception handler is located in a different context, a *Firewall-Interrupt* is triggered. The LR field located in the exception frame indicates a nested exception with the *EXC_RET* value. Based on this information, the location of the exception frame from the second interrupt is known. It is also located on the secure MSP - just before the exception frame of the *Firewall-Interrupt*.

   By applying the offset of the second frame, the PC of the interrupted exception handler is resolved. This address is pushed to the call-stack to identify an exception-return.

3. Based on the configured priority levels, even more layers of nested exceptions can occur. Fortunately, the sequence is always identical from the second layer onwards. The secure main stack is growing with the additional exception frames and local data of the handlers. The call-stack also increases to match the nested call hierarchy.

4. Returning from each layer immediately causes a *Firewall-Interrupt* to activate the initially running context again. The matching address on the call-stack identifies the return sequence. The default exception-return mechanism and the handler code clean-up the main stack without intervention of the context manager.

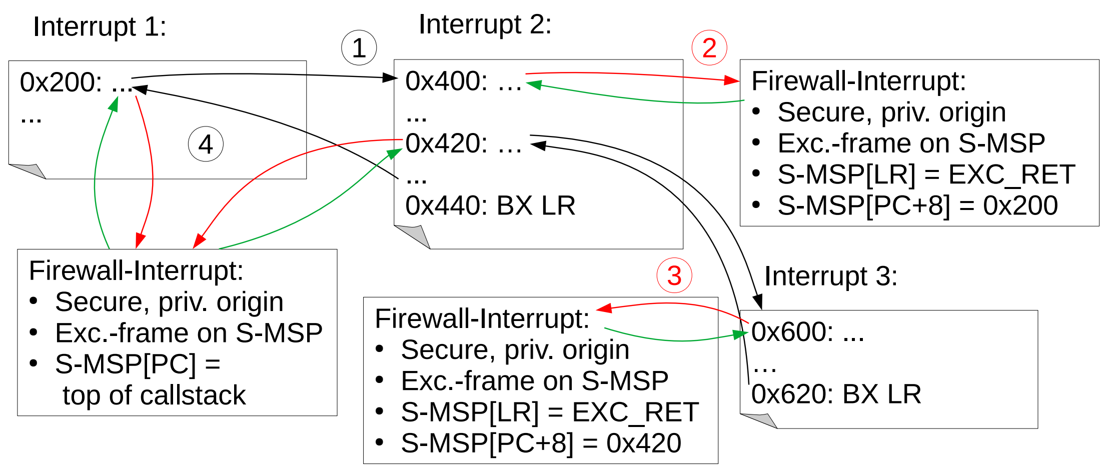

Figure 5.11: Nested exceptions

Summary

By investigating the detailed context transition sequences, the logic of the context manager can be summarized with the fields of Table 5.1: The *EXC_RETURN* value is parsed to get the security domain (S) and execution privilege (Mode: *T*hread or *H*andler) of the interrupt

Table 5.1: Context transition overview

	EXC_RETURN				Return address	Stack frame copy
Fault reason	S	Mode	SPSEL	Call-stack top	source	(bytes)
CALL	S	T	PSP	-	S-PSP[LR]	32
RETURN_S	S	T	PSP	S-PSP[PC]	-	32
RETURN_NS	NS	T	PSP	NS-PSP[PC]	-	0
NS_CALLBACK	NS	T	PSP	-	S-PSP[0]	8
NS_IRQ	NS	H	MSP	-	S-PSP[64]	72
S_IRQ	S	H	MSP	-	S-PSP[PC]	32
S_IRQ_PRIVILEGED	S	H	MSP	-	S-MSP[PC + 8]	0
S_PRIV_CALL	S	H	MSP	-	S-MSP[LR]	0

origin, as well as the location of the exception frame (SPSEL). Additional comparison of the PC to the call-stack is used to identify all return-related sequences.

The location of the return address to the old context is different for every transition. This address needs to be retrieved from the correct stack and offset for updating the call-stack.

Finally, different amounts of data have to be copied when the secure PSP is changed. In the obvious cases (e.g. CALL and RETURN_S), the exception frame of the *Firewall-Interrupt* has to be moved. It is needed immediately after the context manager is finished, to return from the interrupt.

Other cases move data, which is needed later in the return sequence. In the *NS_ CALLBACK* case for example, the exception frame is located on the non-secure stack. The context manager copies the secure return address and status register, which are automatically retrieved from the secure stack when switching the security domain at function return.

5.3.4 Task control block

The responsibility of the Task Control Block (TCB) is storing persistent information when a context is temporarily disabled. Although this project does not specifically deal with RTOS tasks, the name *TCB* is used to follow the naming convention generally used in similar projects. A TCB designed for preemptive scheduling often stores a lot of data from the interrupted task. It can hold general-purpose and control registers, stack information and independent interrupt configuration for every task.

Because of the interrupt-focused design of the context manager, a lot of context data is automatically stored in the exception frame. This enables a lightweight design of the TCB. Only the following entries are stored for every context:

- **Secure PSP:** Because the stack is exclusive for every context, the current address has to be stored before switching to a different stack. When the exception frame of the *Firewall-Interrupt* is moved to the new stack, the address has to be adjusted accordingly.

- **Stack limit:** Both secure stacks have a limit register provided by the core architecture. Modifications of the pointers cause a comparison with the limit and trigger a *UsageFault* if the address underflows.

  They can be optionally used to prevent system corruption by stack underflows. Handling of those cases has to be implemented manually based on the application requirements. Setting the value to zero will lead to the default behaviour which does not restrict the allowed range.

  The limit is stored and set individually for the secure PSP of every context.

- **Switch reason:** In the current implementation, the reason of the last context switch is also stored. This is done for debugging and monitoring purposes. It can be omitted by implementations for productive deployment.

The current implementation does not explicitly store and clear the general-purpose registers when switching between contexts in the secure domain. When implementing this feature, they must not be stored in static memory of the TCB. Instead, they have to be pushed to a stack for the following reason: Context activations can happen interleaved before returning finally. In this scenario, multiple versions of the general-purpose registers exist for a single context. The older version would be overwritten if simply stored in the TCB. Figure 5.12 illustrates this scenario.

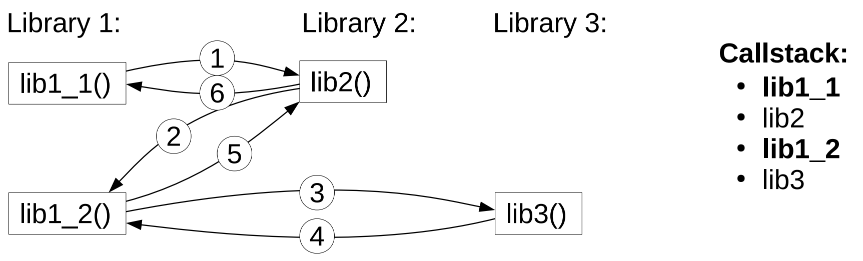

Figure 5.12: Interleaved context switches

Calls of only two contexts can already cause this scenario. Having the same context on the call stack multiple times is caused by a call-back to a function of the source context. If this function again produces a context switch by a function call, the values of the general-purpose registers of the initial call would be lost. For this reason, they have to be stored in a stack data structure. This procedure was not implemented in the scope of this project and left for future work.

5.4 Verification

The two test libraries and the non-secure code are used to perform various use cases. The API functions mostly perform simple arithmetic functions with prime numbers on a given input, and return the result to the caller. This places some arbitrary load on register and stack usage, as well as increasing the code size.

When calling the test functions, the returned results are collected. The current implementation of the context manager features a function to disable it completely. This feature is used, to run the tests again without interference of the firewall. By comparing the returned results, stack or register corruption caused by the context manager can be discovered. Figure 5.13 lists the involved functions for verification.

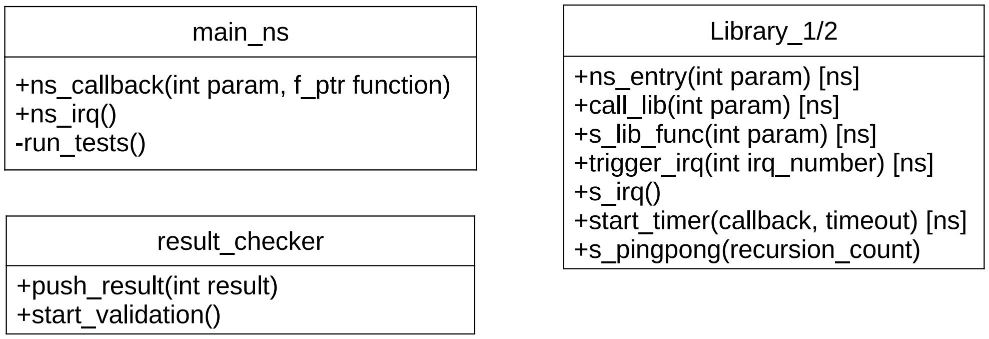
main_ns
+ns_callback(int param, f_ptr function)
+ns_irq()
-run_tests()
result_checker
+push_result(int result)
+start_validation()
Library_1/2
+ns_entry(int param) [ns]
+call_lib(int param) [ns]
+s_lib_func(int param) [ns]
+trigger_irq(int irq_number) [ns]
+s_irq()
+start_timer(callback, timeout) [ns]
+s_pingpong(recursion_count)

Figure 5.13: Verification functions

The executed tests are selected by pre-processor defines. All active tests originate from the non-secure main and the *run_tests* function. The non-secure application also contains an interrupt handler and a call-back function. This call-back can further be used for calling another function.

Most tests are statically implemented and require no modifications for execution. Some more advanced scenarios were tested with temporary code changes. Negative testing is also not performed automated. The resulting *HardFault* halts execution when an invalid sequence was discovered successfully. Both test-libraries contain the same basic functionality to test different ordering of the sequences:

- **ns_entry:** A simple secure entry from non-secure code is tested by calling this function. The passed parameter is modified by the function and returned.
- **call_lib:** This function is used to test a context switch within the secure domain. The secure library function of the other context is called.
- **trigger_irq:** With the *NVIC→STIR* register, an arbitrary interrupt can be triggered by software. It is used by this function, to cause context switches by exceptions in other libraries. The corresponding interrupt hander, implemented in the other context, will be executed.
- **start_timer:** A timer can be started for multiple advanced scenarios. By passing a pointer to a call-back function, a privileged function call is tested. When an interrupt is triggered from this function, tail chaining and nesting of exceptions is performed.
- **pingpong:** This function calls the matching definition in the other context recursively. It is used to test the call-stack management when interleaved switches occur.

Chapter 6

Evaluation

The following sections review the existent design and implementation to highlight potential limitations. It is dependent of the target application if the context manager can be deployed, or if the restrictions prevent the usage for this specific case.

6.1 Timing behaviour

The additional overhead of the context manager causes an unavoidable latency. Variations in the logic and varying copy operations cause differences of the timing behaviour for every fault reason. Measuring of the timings was done by toggling a General Purpose Input/Output (GPIO) pin and capturing the level with an oscilloscope.

6.1.1 Measurement procedure

Additional code was added to set the state of the GPIO pin. This approach introduces a static error in the results, because additional cycles are lost by writing the register values. When a more precise analysis without modification is required, an external trace adapter can be used for monitoring the precise instruction timings.

The pin is toggled before triggering the context switch with a call from a test function. The output voltage starts to increase and reaches a stable level during execution of the subsequent instruction. This steady increase, caused by charging the involved capacitances, is another source for a static error. Because of its small size, it does not influence the resulting cycle count. Nevertheless, the final computation of the *delta time* takes this error into account.

Figure 6.1 classifies the individual contributions to the total latency for a simple function call.

Measurements were performed for three different sections:

- For comparison, the sequence was completed with disabled context manager. This gives a base value for the internal latency caused by instruction fetch/decode and changes in the operation mode. The branching instruction itself, and the instructions

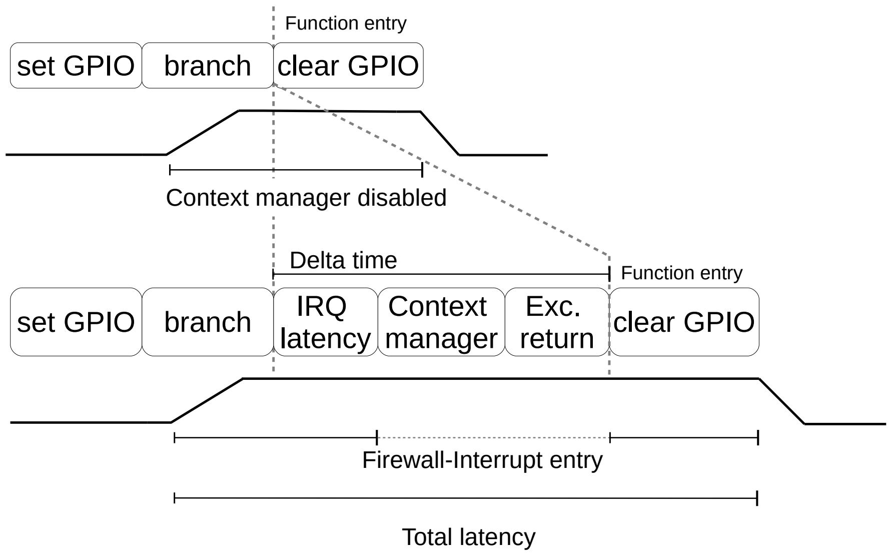

Figure 6.1: Composition of execution latency

to clear the GPIO are also included in this time. This overhead is approximately 1-4 cycles for an unconditional branch, and up to 7 cycles for modifying the GPIO registers [3].

- When the context manager is enabled, fetching the instruction from a disabled memory section causes a *Firewall-Interrupt*. The measured latency includes the failed instruction fetch, triggering of the interrupt and interrupt entry code. This value can be subtracted to get a clearer indication about the isolated context manager performance.
- The total latency caused by the context manager is measured by clearing the pin-value in the target function of the destination context. It defines the time, which is needed before the target function is executed. The instructions for branching and clearing the GPIO are also included in this measurement.
- Finally, the *delta time* is calculated by subtracting the latency with disabled context manager from the total latency. This value is the concrete delay added by the context manager, which has to be accounted for when working with real-time constraints. The added error caused by branching or triggering the context switch (e.g. setting an interrupt request) is compensated by calculating the difference.

6.1.2 Results

The results are only provided to get a general idea of the magnitude of expected delay and define a starting point for comparing later optimizations. Subsequent addition of new features and design changes, as proposed in Section 7.1, will require a new review of the exact status.

With a clock frequency of 100 MHz, the available prototype board has a cycle period of 0,01 μs. With an average time of 4.28 μs spent by the context manager, about 400 cycles are additionally used by the current implementation. Table 6.1 lists the detailed measurement values.

Table 6.1: Context switch latency [μs]

Fault reason	Total latency	Context manager disabled	Interrupt entry	Fault-stack copy [bytes]	Delta time
CALL (NS→S)	4,43	0,15	0,30	32	4,28
CALL (S→S)	4,46	0,11	0,26	32	4,35
RETURN_S	4,30	0,08	0,23	32	4,22
RETURN_NS	4,00	0,16	0,28	0	3,84
NS_CALLBACK	4,60	0,32	0,48	8	4,28
NS_IRQ	5,10	0,38	0,48	72	4,72
S_IRQ	4,90	0,24	0,38	32	4,66
S_IRQ_PRIVILEGED	4,25	0,24	0,38	0	4,01
S_PRIV_CALL	3,77	0,14	0,33	0	3,63

The latency is composed of the following components:

- Security mode change: Switching the security domain is done highly optimized in hardware. Nevertheless, the additional stacking operations cause slow memory operations which add multiple clock cycles of delay. Moreover, the *C*-compiler adds code to clear the general-purpose registers when switching to the non-secure domain. Manually clearing only security critical registers could reduce this proportion slightly. As described in Section 6.1.3, this example takes 4 additional cycles when a change in the security domain is needed.

- Interrupt latency [15]: Interrupt handlers can not be executed without delay for several reasons. Because the proposed architecture is highly dependent on interrupts, this delay is also relevant.

  Completing the current instruction, which might take multiple cycles, adds the first part of the delay. Fetching and decoding the first instruction of the interrupt handler also needs time. Additionally, the instruction pipeline is flushed and needs to be filled again with Interrupt Service Routine (ISR) instructions.

  The minimum interrupt latency for *Cortex-M3* processors is 12 cycles. Measurements for the available *Cortex-M33* prototype showed similar values for the best case. Depending on the source and destination execution mode, a higher latency has to be assumed. Table 6.1 lists the entry delay for the *Firewall-Interrupt* handler. The timing includes the additional cycles for toggling the GPIO pin, which was measured at 0,09 μs (9 clock cycles).

- The largest portion of the latency comes from the logic of the context manager. A decision tree has to be traversed to retrieve the current reason for the switch. The additional sanity check has to look up the entry point in a whitelist when a function call was performed.

Only then, the actual context switch can finally be performed. Based on the switch reason, the exception frame and additional context data has to be moved. Updating the internal data structures and restoring the saved registers also takes additional processor time.

In contrast to the other latencies, this part can be further optimized to increase the responsiveness of the system. Prohibiting some of the more advanced use cases can simplify the decision logic arbitrarily, if they are not used in the application.

The current implementation has the advantage of a simple design with a static address layout. Dynamic registering of libraries and the provided entry-points will increase the complexity of the sanity check and further decrease the overall responsiveness of the context manager. This compromise, whether latency or flexibility is preferred, has to be decided based on the target application.

6.1.3 Latency variations

The different scenarios have a jitter in the latency. One contribution to this variation comes from the different entry times to the context manager. Based on the needed change in execution mode, different preparatory tasks (e.g. stacking operations) are performed by the *ARM* core. The rest of the variation is caused by the fault-reason dependent logic and varying operations handled by the context manager. Figure 6.2 compares the complete delay (Latency total) to the latency introduced purely by the context manager (Delta).

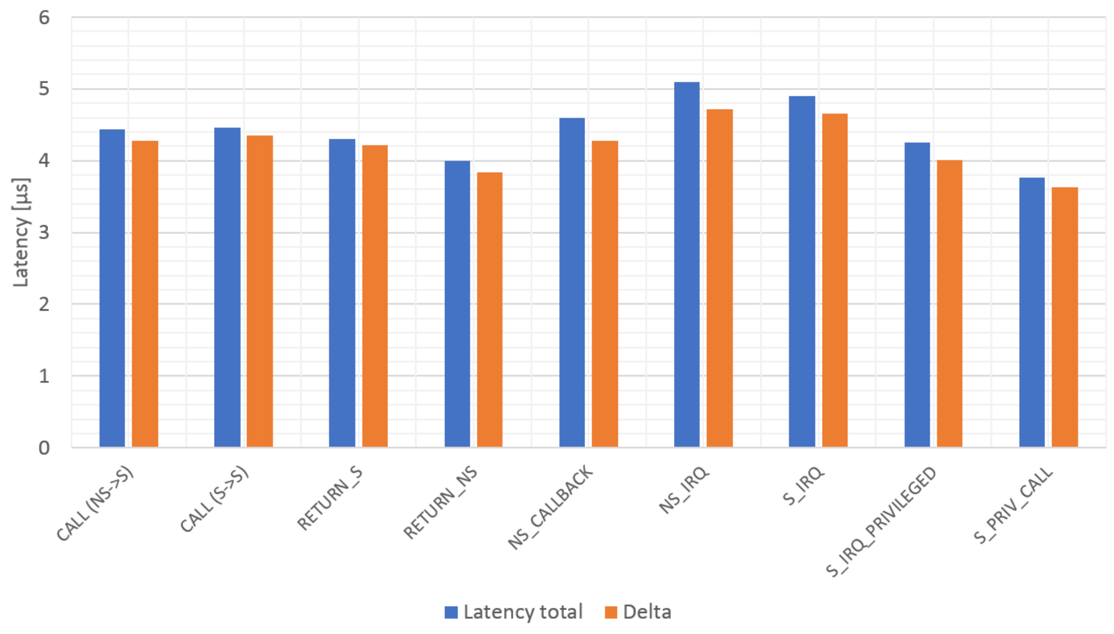

Figure 6.2: Total latency comparison

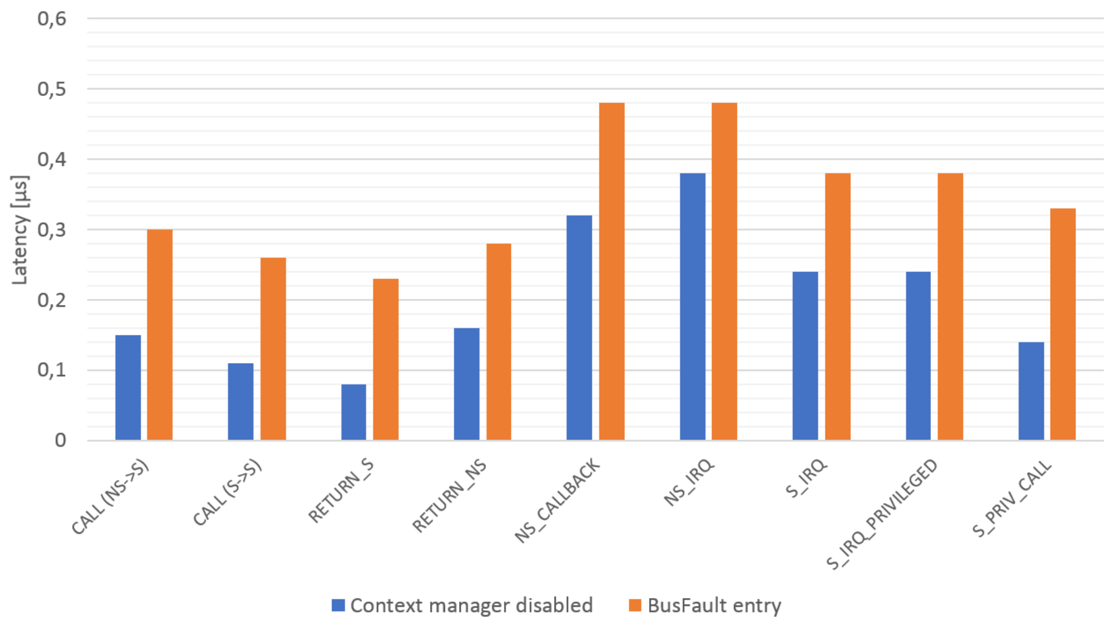

Figure 6.3: Core latency comparison

When looking at the latency of a function call to the secure domain, a higher delay is observable when the *TrustZone* border is crossed. This factor is apparent when looking at the timings with disabled context manager. Compared to calls within the same security domain, four additional cycles are needed. This latency is also present when looking at the time needed for the *Firewall-Interrupt* entry. These two scenarios with their respective timings are shown in Figure 6.3

Returning to the non-secure domain generally takes more time. This is caused by the additional clearing of the general-purpose registers. With the activated context manager on the other hand, copying the exception frame for the secure destination and the additional logic outweighs this small delay substantially.

The most internal latency of the core is detected, when non-secure code has to return to the secure domain. This is the case for non-secure interrupts and call-backs. A non-secure interrupt also requires moving of the large additional state context. This causes a large latency and makes it the least responsive case. Delaying interrupts by such a long time might be a critical factor. At least for non-secure interrupts, this can be solved by placing them in an always-on section.

Secure interrupts have a lower entry delay compared to the non-secure counterparts, but the internal logic is more complicated. With the current implementation, checking for tail-chained exceptions requires more operations than a standard call-sequence. Interrupts during privileged execution have the same entry delay, but return to normal execution more quickly. The reduced total time results from omitting exception frame movement.

As for the non-secure case, this interrupt delay could cause problems for some applications. This issue can also be solved by defining a section, which is always active and not assigned to a context. Special care has to be taken when placing secure code in such an always-on section: Data can be leaked, and code might be injected when the section is not configured as execute-only.

Calling a secure function from privileged execution is the most responsive sequence. No switch in execution privilege or security state is needed. There is also no copy operation for moving the exception frame required.

6.2 Attack vectors

The responsibility concerning system security is shared between the *TrustZone* and the context manager with hardware support from the *AHB-Firewall*. Nevertheless, the compiler toolchain and the programmer must also produce security aware bytecode.

The following section lists potential attack vectors and describes the applied solution defending against it. A short summary repeats the main points of the affected logic.

The initial attribution of memory security of the target hardware is done with the provided IDAU. The current hardware implementation assigns the security status in blocks of 4 kB, which might waste memory space in some instances. This limitation is especially severe for NSC sections. It is highly unlikely, that the secure veneers occupy a major portion of this section.

The remaining space is lost and must be reserved by the linker script to prevent accidental placement of code or data. This is an essential requirement to protect against invalid entry to the secure domain and arbitrary code injection. Moreover, the placement address used during the linking stage must match the address configured by the code. Because the final location is dependent on the size of preceding code, it might be moved during development. This is a chicken-egg-problem which can be solved by using linker symbols for the IDAU configuration.

The integrity of the configuration values and code has to be ensured with secure ROM or cryptographically signed firmware binaries. The setting also has to be applied as soon as possible and measures against reconfiguration have to be put in place.

When the attribution is configured correctly, the basic partitioning into secure and non-secure domains is guaranteed by the *TrustZone*. Generally speaking, this statement is only valid for the software execution on the *ARM*-core. Security attribution for other components of the SoC attached to the system bus is highly (hardware-)implementation dependent. In the worst case, all bus members are configured with secure authority. This configuration would not increase the resilience against external threads compared to a MPU-based security architecture (see Section 2.1.1 for details).

With this system in place, many potential attack vectors are already covered by the hardware implementation. One of the most basic scenarios, a function call from the non-secure domain to a secure API, is handled by the *TrustZone* at the first stage. Accesses to secure memory are prohibited, only instruction fetches to NSC memory are permitted when the SG operation is the target.

Nevertheless, there are still possibilities for security violations not covered by this hardware logic: The *C* compiler must add code to clear the general-purpose registers before returning to non-secure memory. When using the assembler language directly, it is the responsibility of the programmer to prevent leakage of confidential data.

Without context manager, secure code can access the complete memory map. Non-secure code might wrongfully supply secure pointers to cause unchecked memory corruption. The destination for modifications has to be validated by the secure code for this reason. With the context manager, access to memory of other contexts is prevented. Data has to be placed in shared buffers located in an always-on section beforehand.

Non-secure call-backs are supported directly by the hardware. The secure return address is automatically pushed to a secure stack, to prevent the non-secure code from returning to an arbitrary address. The return sequence is also fully enabled by the hardware. Clearing confidential data from registers is the responsibility of the compiler and programmer. Call-backs are generally not regulated by the *TrustZone* technology. Branching to arbitrary non-secure addresses is possible. The context manager on the other hand, could be used to also restrict those transitions to predefined entry functions.

Once the secure domain is entered, no further monitoring is performed by the *TrustZone* technology. Apart from privileged configuration registers, the complete memory map is accessible by default. Data can be modified, and functions can be called arbitrarily.

Distinct firewall sections are used by the context manager to limit this unrestricted behaviour. The additional sanity check prevents calling of arbitrary functions outside the active context. By detecting crossing of context borders, the context manager can clear potentially secure data from the general-purpose registers. The current implementation does not allow data access to the memory of another context. Shared buffers in always-on regions have to be used for passing large amounts of data.

The sequence of non-secure exceptions, interrupting secure code is fully handled by the build-in *TrustZone* hardware. Possibly confidential data is automatically pushed to the secure stack before switching to the non-secure execution state. The well-proven exception-return sequence ensures that execution continues at the correct address. The only software interaction happens, when the exception handler is placed in an inactive context. The interrupted context is deactivated during execution of the handler, to protect it from malicious operations.

Secure interrupts use the default exception sequence. The only addition is the change in execution mode, when needed. This simple mechanism can be used, because data leakage from the non-secure domain or other secure code is not considered critically by the initial *TrustZone* design. However, before returning, the compiler toolchain or the programmer have to clear the general-purpose registers to prevent security violations. This responsibility can also be moved to the context manager when a switch is performed.

Table 6.2 gives a summary of the different attack vectors and the corresponding module defending against it.

Table 6.2: Attack vector summary

Use case	**Task**	**Responsible module**
NS→S call	Block calls to arbitrary addresses	TrustZone
	Clear confidential registers before returning	Compiler/Programmer
	Check access permission of passed pointers	Programmer
NS→S data access	Block all accesses	TrustZone
NS callback	Change security state and banked registers	TrustZone
	Block calls to arbitrary addresses	Firewall/Context manager
	Clear/restore confidential registers	Compiler/Programmer
	Ensure return to correct secure address	TrustZone
S→S call	Block calls to arbitrary addresses	Firewall
	Check correct entry/perform context switch	Context manager
	Clear/restore confidential registers	Context manager
S→S data access	Block all accesses (current implementation)	Firewall/Context manager
NS interrupt	Change security state and banked registers	TrustZone
	Clear/restore confidential registers	TrustZone
	Ensure return to correct secure address	TrustZone
	Context switch	Firewall/Context manager
S interrupt	Change security state and banked registers	TrustZone
	Clear confidential registers	Compiler/Programmer

Chapter 7

Conclusion

This project examined the security features of the new *ARMv8-M* architecture and an additional proprietary AHB firewall. With the practical use case of a payment terminal and the challenges originating from the IoT market, the importance of software security in embedded systems was emphasized.

The design of already established RTOS projects was evaluated, to get ideas and technical background information for designing a system for secure context isolation.

A memory partitioning was created and the details of the various transition sequences were tested to cover all required use cases. The implementation acts as proof-of-concept and can be used to further improve the design. It also provides a basis for measuring the expected latency and evaluating the architecture for practical use cases.

Additional features, which were not implemented in this project, are mentioned in the next section. It also includes recommendations for increasing the performance and flexibility of the design.

In conclusion, the design and the resulting implementation satisfy the initial requirements: With the help of the *TrustZone* and the AHB-firewall, it is possible to protect secure code and data from unauthorized accesses. Debug access is configured individually for every section, providing essential developing support for third-party customers.

The implicit context switching, realized by handling the *Firewall-Interrupt*, enables flexible deployment of the project: Application code is simply linked against the supplied import library of the secure API and loaded for execution. No RPCs or manual context switching by calling special functions is required. This allows easy porting of legacy projects and developing a single code-base for products with and without the integrated context manager.

The additional time of the context switch is in the same range as task switches performed by established RTOS projects. This should not represent a knock-out criterion for most applications. Whether interrupt handlers can also be protected, depends on the individual performance requirements concerning latency. As a trade-off, they could be placed in shared regions.

7.1 Future work

The presented project is only concerned with the software part of the context switching and processing of the multiple transition sequences. It acts as proof-of-concept implementation for the underlying design. There are still refinements needed before deploying it to a productive system. Additionally, it needs to be integrated with supporting security concepts to act as a complete base for a RoT solution.

The following summarizes key requirements, to provide the intended functionality and optional modifications to increase performance and overall usability.

7.1.1 External requirements

Additional support from external components is required to ensure system security. To provide the basic functionality of the proposed design, one critical change has to be made at the AHB-firewall: The current implementation does not distinguish between privileged and unprivileged execution modes.

As a measure against unauthorized modifications, the configuration registers of the firewall can be locked-down completely. However, this would also prevent the dynamic reconfiguration performed by the context manager. Defining a firewall section, which covers the address range of those memory-mapped registers, can be used as a workaround. This prevents untrusted code in the non-secure domain from disabling the firewall, but still gives all secure code the authority for modifications. The isolation of secure contexts cannot be guaranteed by this approach - it is only applicable when the complete secure code is part of the TCB.

A proposed design improvement would only allow secure, privileged code to access those registers. The protection of MPU configuration and some system registers against unprivileged tampering follows the same idea.

Ensuring confidentiality during execution is only a partial solution to the underlying problem. Third-parties must be able to efficiently develop application code without gaining access to the internals of the IP. Integrity of configuration and code has to be guaranteed to prevent subsequent modifications. This requires encrypted binaries which are processed by a secure bootloader. The non-secure application is then statically linked to the secure APIs and also loaded at start-up.

Based on the memory technology of the target hardware, performance of context switching could be further improved. By placing the context manager in Tightly Coupled Memory (TCM), the latency of instruction fetches can be reduced in comparison to flash or normal RAM [34]. This is possible, because the memory is directly attached to the core and runs at the same clock to provide single-cycle access.

Another hardware-based performance increase could be achieved by implementing some of the decision logic in hardware. The algorithm to decide the fault reason consists mostly of Boolean comparison operations. With a code size of nearly 280 bytes, traversing this decision tree adds unnecessary latency.

7.1.2 Implementation changes

The current implementation focuses on the core logic and verifying the processing of various domain transitions. To simplify the first version of implementation, only uniform sections, located on static address ranges are supported. This is not a practical solution, because the sizes of different contexts might be largely different. The address ranges should also be computed dynamically based on the starting point used during linking.

The target hardware platform does not include an FPU, simplifying the handling of the fault stacks. When a hardware unit is present, additional registers might be stacked at exception entry. This is indicated by the *FType* field of the *EXC_RETURN* value and the *FPCCR.TS* field. The current implementation does not consider this additional floating-point context when moving the exception frame.

The callee-saved registers are not saved by the current implementation, because the ABI requires subroutines to restore them before returning. To provide full protection against malicious attacks or malfunctioning code, all register have to be stored when switching to a different context. The contents should also be cleared to prevent leakage of secure data. This protection was not implemented for the current version, because secure code is expected to comply to the ABI.

Another simplification was introduced by limiting the effective range of the context manager to secure memory. All non-secure memory is combined into a single security domain with full access. It is common practice to isolate the RTOS kernel from application tasks by using the MPU. On the used hardware platform, the MPU was omitted in favour of the proprietary AHB-firewall. This circumstance makes isolating the kernel from application code impossible at the moment. Extending the context manager by the missing transition sequences needed for the non-secure domain, could provide a solution for this hardware limitation.

The context manager is currently tightly coupled to the interface of the AHB-firewall. Changes in behaviour of the hardware module have to be also replicated in this code. A Hardware Abstraction Layer (HAL) should be introduced to encapsulate the firewall-specific functions from the context manager. This would also enable the usage of other memory protection schemes in the future. Based on preferences, the MPU might be used for context isolation on other hardware platforms, instead of the firewall.

To have a functional and secure system, the configuration of three different components must match: Firstly, the linker scripts of both security domains are responsible for placing the code and data in defined sections. The security attribution is then configured with the IDAU settings. This provides the underlying partitioning in secure and non-secure domains enforced by the *TrustZone*. Only then, the AHB-firewall is used for more detailed attribution based on access type and type of bus master.

If the memory map of the target application is not completely fixed during the design stage, address ranges can change during implementation. This happens when relative addresses are used in the linker script and additional components are inserted at the beginning of the memory map. Another cause of this phenomenon is increasing buffer sizes or changing the optimization level. In the best case, the linker complains about range violations or the application crashes immediately. Hard to debug faults can occur, when only certain functions or segments of buffers are shifted to invalid sections.

To prevent the mentioned misconfiguration risk, linker symbols and configuration functions should be introduced to dynamically set the section ranges.

The current implementation uniquely assigns memory sections to contexts. This prevents the possibility for shared buffers, where only a subset of contexts has access rights. To further extend this scenario, every context could have different access rights for a single section: A possible application for this feature would be a data sink, where one context has only write permissions and acts as data producer (e.g. receiver interface), and a different context is the consumer which processes the data (e.g. decrypting data) with full read/write access.

This feature could also be a prerequisite for some interrupt handlers in always-on regions: The handlers for real-time critical interrupts can be located in an execute-only section which are always enabled, to avoid the latency of the context manager. They can set events to trigger the task scheduling or modify other internal states.

If security critical data is involved, special precautions have to be in place. Previously mentioned data-sinks have to be used instead of public buffers. Writing to them does not cause a context switch and only the target context has read privileges.

When the stack-limit registers are used, the expected maximum stack size should be independently configurable for every context. Otherwise, contexts with simple functions, that only use little stack memory would waste a lot of memory when only a global stack size is used.

As a final statement, the following principle should be considered for extending the context manager: The fact must be accepted, that added security results in performance losses and restricts the flexibility of the platform. It is a challenging task to find the best compromise without ending up in a cycle as depicted in Figure 7.1.

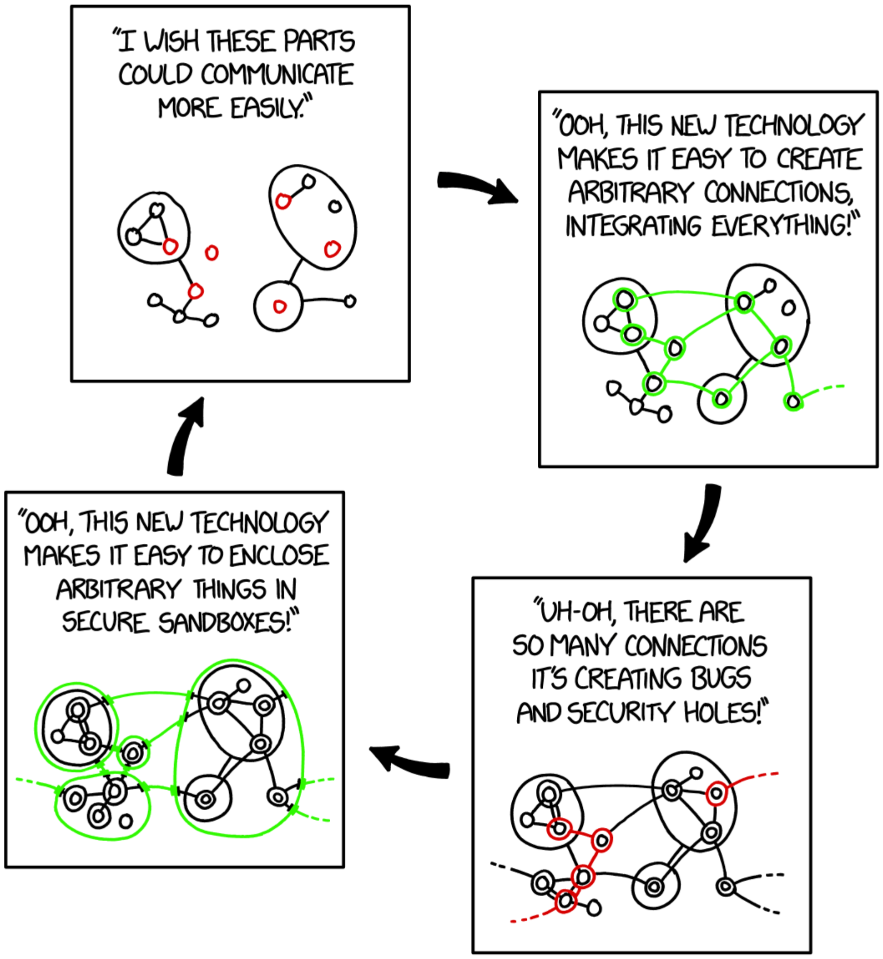
"I WISH THESE PARTS COULD COMMUNICATE MORE EASILY."
"OOH, THIS NEW TECHNOLOGY MAKES IT EASY TO CREATE ARBITRARY CONNECTIONS, INTEGRATING EVERYTHING!"
"UH-OH, THERE ARE SO MANY CONNECTIONS IT'S CREATING BUGS AND SECURITY HOLES!"
"OOH, THIS NEW TECHNOLOGY MAKES IT EASY TO ENCLOSE ARBITRARY THINGS IN SECURE SANDBOXES!"

Figure 7.1: Sandboxing Cycle [17]

Terms and abbreviations

ABI	Application Binary Interface 38, 84
ACL	Access Control List 31
AHB	Advanced High-performance Bus 12, 39–41, 79, 82–84
API	Application Programming Interface 10, 12, 20, 24, 26, 29–31, 34, 36–38, 41, 61, 72, 79, 82, 83
AWS	Amazon Web Services 30
BXNS	Branch and Exchange Non-secure 21
CLIF	Contactless Interface 13, 41
DDoS	Distributed Denial of Service 11
DMA	Direct Memory Access 16, 39, 40
DRM	Digital Rights Management 33
DSP	Digital Signal Processor 39
DVR	Digital Video Recorder 11
FPGA	Field Programmable Gate Array 39
FPU	Floating-Point Unit 30, 84
GCC	GNU Compiler Collection 57
GPIO	General Purpose Input/Output 74–76
GUI	Graphical User Interface 12, 27
HAL	Hardware Abstraction Layer 84
I^2C	Inter-Integrated Circuit 39
IDAU	Implementation Defined Attribution Unit 18, 39, 40, 79, 84
IDE	Integrated development environment 57
IoT	Internet of Things 2, 3, 10, 11, 13, 14, 30, 31, 35, 37, 82

IP	Intellectual Property 10–12, 24, 38, 58, 83
IPC	Inter-process communication 36
IPSR	Interrupt Program Status Register 21, 44
ISR	Interrupt Service Routine 76
kB	kilo-byte 39, 40, 42, 44, 79
LR	Link Register 21, 22, 45, 60, 61, 64–67, 69, 70
MMU	Memory Management Unit 2, 3, 25, 28, 33, 35
MPU	Memory Protection Unit 9, 13, 15, 16, 18–20, 28–30, 37, 79, 83, 84
MSP	Main Stack Pointer 19, 44, 50–53, 66, 67, 69, 70
NFC	Near Field Communication 12, 26
NMI	Non-maskable Interrupt 43
NSC	Non-Secure Callable 17, 20, 44, 47, 58, 60, 79
NVIC_ITNS	Interrupt Target Non-secure 21
PAN	Primary Account Number 27
PC	Program Counter 66, 67, 69, 70
PCI	Payment Card Industry 27
PendSV	Pended Service Call 30, 52, 53
PIN	Personal Identification Number 26, 27
PLC	Programmable Logic Controller 11
POS	Point of Sale 7, 12–14, 26
PSA	Platform Security Architecture 35
PSP	Process Stack Pointer 19, 30, 44, 46–48, 53, 61–64, 66, 67, 71, 72
RAM	Random Access Memory 15, 33, 39, 83
RO	Read-Only 20
ROM	Read-Only Memory 33, 39, 41, 79
RoT	Root-of-Trust 41, 83
RPC	Remote Procedure Call 12, 25, 38, 82
RTOS	Real Time Operating System 12–16, 19, 28–30, 36–38, 41, 71, 82, 84
SAU	Secure Attribution Unit 18, 39
SG	Secure Gateway 20, 44, 47, 60, 79

SMC	Secure Monitor Call 25, 33, 34
SoC	System-on-Chip 7, 10–12, 15–17, 19, 26, 33, 39, 79
SPI	Serial Peripheral Interface 39, 41
SVC	Supervisor Call 31, 34
TCB	Trusted Code Base 3, 27, 34, 37, 38, 83
TCB	Task Control Block 45, 61, 71, 72
TCM	Tightly Coupled Memory 83
TF-M	Trusted Firmware-M 35, 37
UART	Universal Asynchronous Receiver Transmitter 39
VTOR	Vector Table Offset Register 19

Bibliography

[1] ARM. *ARM mbed Technical Overview*. Web. Accessed: 13.10.2018. June 2017. URL: `https://www.arm.com/files/event/20170628_ATF_Korea_B2.pdf`.

[2] ARM. *The Arm Mbed uVisor*. Web. Accessed: 13.10.2018. 2018. URL: `https://github.com/ARMmbed/uvisor/blob/master/README.md`.

[3] *ARM Cortex-M3 Processor Technical Reference Manual*. r2p1. Accessed: 14.10.2018. ARM. Feb. 2015. URL: `http://infocenter.arm.com/help/topic/com.arm.doc.100165_0201_00_en/arm_cortexm3_processor_trm_100165_0201_00_en.pdf`.

[4] *ARM®v8-M Architecture Reference Manual*. Beta A.b. ARM. July 2016.

[5] *ARMv8-M Memory Protection Unit*. Version 2.0. Accessed: 01.10.2018. ARM. 2017. URL: `https://static.docs.arm.com/100699/0200/armv8m_memory_protection_unit_100699_0200_en.pdf`.

[6] Aspencore, EETimes, and Embedded.com. *2017 Embedded Markets Study*. Web. Accessed: 09.10.2018. Apr. 2017. URL: `https://m.eet.com/media/1246048/2017-embedded-market-study.pdf`.

[7] *embOS-MPU - Real-Time Operating System User and Reference Guide*. Revision: 1. SEGGER. Dec. 2017. URL: `https://www.segger.com/downloads/embos/UM01001`.

[8] Andreas Fitzek. "Development of an ARM TrustZone aware operating system ANDIX OS". Accessed: 16.10.2018. MA thesis. Graz University of Technology, Apr. 2014. URL: `https://online.tugraz.at/tug_online/voe_main2.getvolltext?pCurrPk=77938`.

[9] *Full TrustZone exploit for MSM8974*. Web. Accessed: 25.09.2018. Aug. 2015. URL: `http://bits-please.blogspot.com/2015/08/full-trustzone-exploit-for-msm8974.html`.

[10] Zhichao Hua et al. "vTZ: Virtualizing ARM TrustZone". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 541–556. ISBN: 978-1-931971-40-9. URL: `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/hua`.

[11] *Introduction to the ARMv8-M architecture*. Version 2.0. Accessed: 04.10.2018. ARM. 2017. URL: `https://static.docs.arm.com/100688/0200/introduction_to_armv8m_architecture_100688_0200_en.pdf`.

[12] C. Kolias et al. "DDoS in the IoT: Mirai and Other Botnets". In: *Computer* 50.7 (2017), pp. 80–84. ISSN: 0018-9162. DOI: `10.1109/MC.2017.201`.

[13] R. Langner. “Stuxnet: Dissecting a Cyberwarfare Weapon”. In: *IEEE Security Privacy* 9.3 (May 2011), pp. 49–51. ISSN: 1540-7993. DOI: `10.1109/MSP.2011.67`.

[14] Trevor Martin. *The Designer’s Guide to the Cortex-M Processor Family*. Elsevier Science, June 6, 2016. 490 pp. URL: `https://www.ebook.de/de/product/26315737/trevor_martin_the_designer_s_guide_to_the_cortex_m_processor_family.html`.

[15] *Measuring Interrupt Latency*. Rev. 1. Accessed: 12.11.2018. NXP Semiconductors. Apr. 2018. URL: `https://www.nxp.com/docs/en/application-note/AN12078.pdf`.

[16] *Memory Protection Unit (MPU)*. 1.0. Accessed: 28.09.2018. ARM. 2016. URL: `https://static.docs.arm.com/100699/0100/armv8m_architecture_memory_protection_unit_100699_0100_00_en.pdf`.

[17] Randall Patrick Munroe. *Sandboxing Cycle*. Web. Accessed: 27.11.2018. Sept. 2018. URL: `https://xkcd.com/2044/`.

[18] *Official FreeRTOS Ports*. Web. Accessed: 09.10.2018. URL: `https://www.freertos.org/RTOS_ports.html`.

[19] *Platform Security Architecture Overview*. Revision 1.2. Accessed: 20.11.2018. ARM. Oct. 2018. URL: `https://pages.arm.com/PSA-Building-a-secure-IoT.html`.

[20] *Programmer’s Guide for ARMv8-A*. Version 1.0. ARM. 2015. URL: `https://static.docs.arm.com/den0024/a/DEN0024A_v8_architecture_PG.pdf`.

[21] *ProvenCore-M Product Page*. Web. Accessed: 21.11.2018.

[22] Dan Rosenberg. “QSEE TrustZone Kernel Integer Overflow Vulnerability”. In: *Black Hat Conference*. Accessed: 25.09.2018. 2014. URL: `www.blackhat.com/docs/us-14/materials/us-14-Rosenberg-Reflections-On-Trusting-TrustZone-WP.pdf`.

[23] Segger. *Basic concepts of embOS-MPU*. Web. Accessed: 19.10.2018.

[24] Segger. *IoT—Smart Embedded Solutions*. Web. Accessed: 10.10.2018. URL: `https://www.segger.com/products/security-iot/iot-solutions/`.

[25] *SMC CALLING CONVENTION*. Issue B. Accessed: 08.10.2018. ARM. 2016. URL: `http://infocenter.arm.com/help/topic/com.arm.doc.den0028b/ARM_DEN0028B_SMC_Calling_Convention.pdf`.

[26] Saleh Soltan, Prateek Mittal, and H. Vincent Poor. “BlackIoT: IoT Botnet of High Wattage Devices Can Disrupt the Power Grid”. In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 15–32. ISBN: 978-1-931971-46-1. URL: `https://www.usenix.org/conference/usenixsecurity18/presentation/soltan`.

[27] D. Suciu and R. Sion. “Droidsentry: Efficient Code Integrity and Control Flow Verification on TrustZone Devices”. In: *2017 21st International Conference on Control Systems and Computer Science (CSCS)*. May 2017, pp. 156–158. DOI: `10.1109/CSCS.2017.28`.

[28] *TEE Client API Specification*. Version 0.17. Accessed: 08.10.2018. GlobalPlatform. 2010. URL: `http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=2C3A19C13215CE2ACB0A0069B09B5039?doi=10.1.1.183.2049&rep=rep1&type=pdf&usg=AOvVaw2JQhCFbWENvqm0t0a6PhRR`.

[29] *The FreeRTOS^{TM} Reference Manual*. Version 10.0.0 issue 1. Accessed: 10.10.2018. Amazon Web Services. 2017. URL: `https://www.freertos.org/Documentation/FreeRTOS_Reference_Manual_V10.0.0.pdf`.

[30] *Using the STM32F0xx DMA controller*. Accessed: 02.10.2018. STMicroelectronics. May 2012. URL: `https://www.st.com/resource/en/application_note/dm00053400.pdf`.

[31] *Using TrustZone on ARMv8-M*. AN291. Accessed: 30.10.2018. ARM-KEIL. 2016. URL: `www.keil.com/appnotes/files/apnt_291.pdf`.

[32] Nathanael R. Weidler et al. "Return-Oriented Programming on a Cortex-M Processor". In: *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE, Aug. 2017. DOI: `10.1109/trustcom/bigdatase/icess.2017.318`.

[33] *Which ARM Cortex Core Is Right for Your Application: A, R or M?* Accessed: 25.09.2018. Silicon Labs. URL: `https://www.silabs.com/documents/public/white-papers/Which-ARM-Cortex-Core-Is-Right-for-Your-Application.pdf`.

[34] Jacko Wilbrink and Lionel Perdigon. "Run Blazingly Fast Algorithms with Cortex-M7 Tightly Coupled Memories". In: (Nov. 2015). Accessed: 16.10.2018. URL: `http://itersnews.com/wp-content/uploads/experts/2015/11/101291Atmel-45151-Fast-Algorithms-with-Cortex-M7_Article.pdf`.

[35] Joseph Yiu. *ARMv8-M Architecture Technical Overview*. Tech. rep. Accessed: 04.10.2018. ARM, Nov. 2015. URL: `https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-2142-00-00-00-00-66-90/Whitepaper-_2D00_-ARMv8_2D00_M-Architecture-Technical-Overview.pdf`.

[36] Joseph Yiu. "Software Development in ARMv8-M Architecture". In: *Embedded World 2017*. Accessed: 07.10.2018. 2017. URL: `https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-2142-00-00-00-01-27-19/ARM-Cortex-_2D00_-session-11-_2D00_-Yiu-_2D00_-Software-Development-in-ARMv8_2D00_M-Architecture.pdf`.