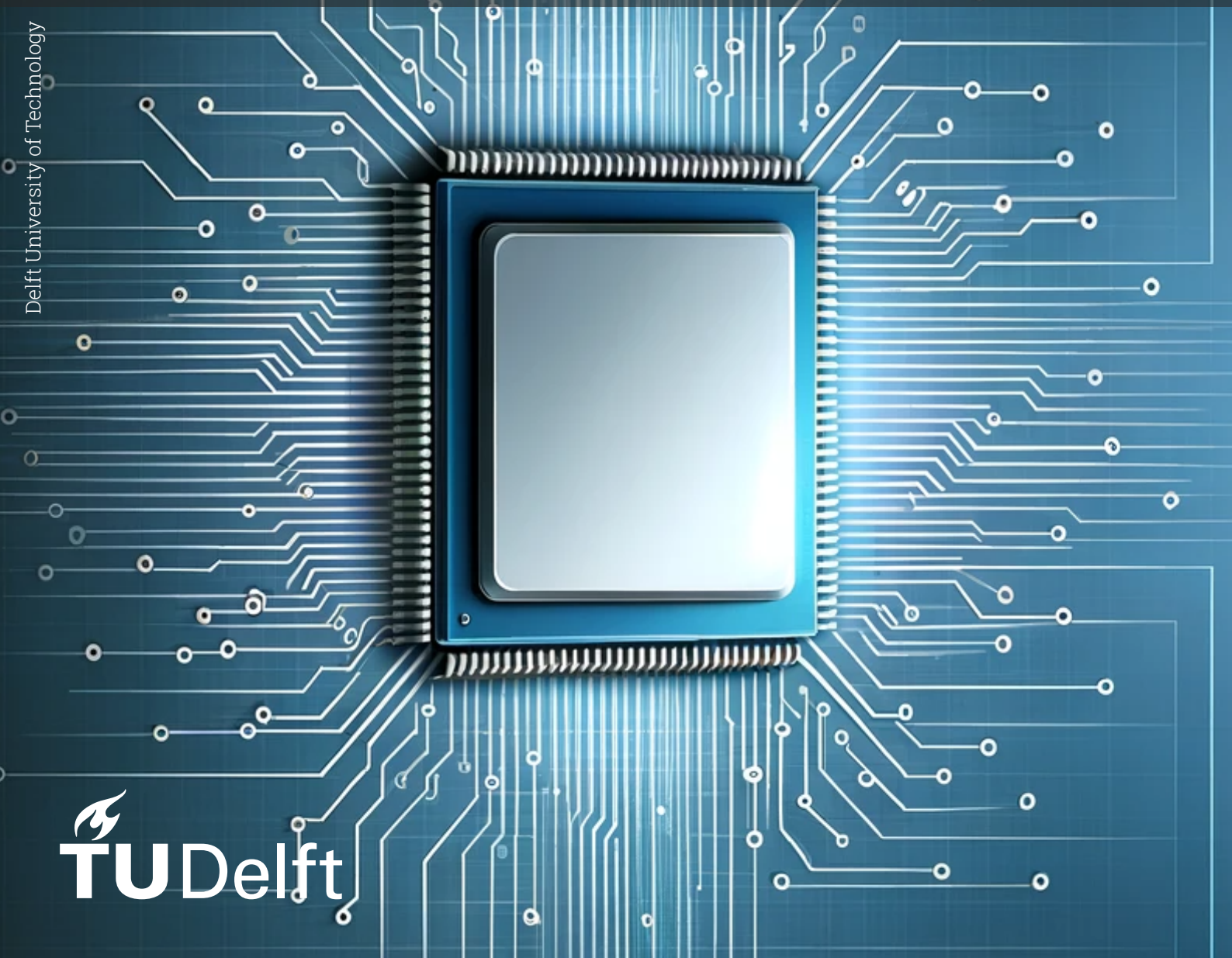
Secure Task Management in FreeRTOS

A RISC-V Core Approach with Physical Memory Protection

MSc Thesis

Christian Larmann



Delft University of Technology

TU Delft

Secure Task Management in FreeRTOS

A RISC-V Core Approach with Physical Memory Protection

by

Christian Larmann

Thesis Committee:	Prof. Dr. Ir. Georgi Gaydadjiev
	Dr. Ir. Mottaqiallah Taouil
	Prof. Dr. Koen Langendoen
Project Duration:	December 2023 - August 2024
Faculty:	Faculty of Electrical Engineering, Mathematics & Computer Science, Delft

Cover:	Image created by DALL·E 3
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

Abstract

The demand for embedded devices, such as intelligent sensors, smartwatches, medical implants, and computing chips in cars, has been rising steadily in the past and is expected to continue for the coming decade. Their high and increasing degree of connectivity expands the attack surface for attackers, which is exaggerated by their physical accessibility. This makes them particularly vulnerable to invasive attacks. Compounding these risks is the growing reliance on third-party software. An example of such risks can be seen in the recent incident where a software solution from the company CrowdStrike, integrated into the Windows Operating System (OS), caused a major global outage that left many airports, hospitals, banks, and government departments worldwide unable to operate. The Guardian reports that this incident affected 8.5 million Windows machines and resulted in an estimated financial loss of 5.4 billion USD for companies in the US alone. Similar risks exist for OSs for embedded devices. These devices do not run general-purpose OSs like Windows, however, in their very nature they are similar. They provide a runtime abstraction for user processes and manage resources for which the OS requires higher privileges. FreeRTOS is a popular embedded OS for resource-constrained devices, also offering real-time capabilities, making it a Real-Time Operating System (RTOS). Meanwhile, RISC-V, a relatively new Instruction Set Architecture (ISA), is gaining popularity due to its open-source philosophy. However, despite the increasing popularity of RISC-V and the widespread use of FreeRTOS, there is a lack of comprehensive security support for FreeRTOS on RISC-V platforms.

This thesis addresses the lack of security for malicious third-party applications and presents a novel FreeRTOS implementation method on a RISC-V embedded system platform. The security of FreeRTOS is enhanced by leveraging the Physical Memory Protection (PMP) feature of RISC-V. The primary contribution involves the FreeRTOS kernel dynamically manage the access rights using PMP, preventing malicious tasks from causing harm. This dramatically improves the security, because the flat memory structure inherent to FreeRTOS is prone to being readily exploited and compromised by attackers. Further, an overview of known attacks on embedded systems and countermeasures is presented, a subset of which are explored in more detail. A secure platform is proposed that aims to demonstrate a holistic approach to integrate the FreeRTOS with PMP support into a System-on-Chip (SoC). Then, the FreeRTOS platform is augmented by a secure boot procedure and a memory encryption unit that ensures the integrity and confidentiality of the external memory. This encryption unit uses the Prince encryption scheme and is elegantly combined with RISC-V's PMP functionality. Unused bits in the PMP configuration registers are used to control the encryption selectively for single PMP memory regions. Moreover, an address-based encryption mode for this unit is proposed that addresses the weakness that plaintext data is always encrypted to the same ciphertext. This is problematic because an attacker could analyze the structure of the ciphertext and copy the ciphertext of memory cells with known plaintext. The effectiveness of the PMP task isolation is demonstrated by simulating malicious tasks that try to access specific data addresses in the memory of other tasks and the kernel. This setup imitates real-life attack scenarios, assuming the addresses have been exposed to the malicious task through reverse-engineering of the application code. The secure boot is validated by implementing the design on an FPGA and manually tampering the memory, resulting in a failed boot-up. Additionally, the impact of the PMP integration on the real-time capabilities of FreeRTOS is investigated more thoroughly by evaluating real-time metrics: task switch time, access time for shared resources using a mutex (a software construct to avoid access collisions), and dynamic memory allocation. The system design incorporates a RISC-V core named CV32E40S, an open-source core from OpenHW Group. All the security enhancements were run on hardware with a varying number of active PMP regions. In the worst-case scenario, using 64 PMP regions results in a 2.1x increase in task switch time, a 2.4x increase in the time to access shared resources using a mutex, and a 1.7x increase in memory allocation time. The area of the CV32E40S, measured in Look-Up-Tables (LUTs), increases with the number of regions: a 2.2x increase for 16 regions, 3.4x for 32 regions, and 5.4x for 64 regions, compared to the core configuration without PMP. Despite these increases, it remains within an acceptable range relative to the total area of the SoC including cache memory and peripherals.

Contents

List of Figures

List of Tables

Abbreviations

AHB Advanced High-Performance Bus
AMBA Advanced Microcontroller Bus Architecture
APB Advanced Peripheral Bus
API Application Programming Interface
ASIC Application Specific Integrated Circuit
BOF Buffer Overflow
CG Cybersecurity Goal
CPU Central Processing Unit
CSR Control and Status Registers
DoS Denial of Service
DDoS Distributed Denial of Service
DS Damage Scenario
FPGA Field Programmable Gate Array
GCC GNU Compiler Collection
IoT Internet of Things
IP Intellectual Property
ISA Instruction Set Architecture
JTAG Joint Test Action Group
LSU Load-Store-Unit
LUT Look Up Table
MAC Message Authentication Code
MCU Microcontroller Unit
M-Mode Machine-Mode
MMU Memory Management Unit
MPU Memory Protection Unit
MSB Most Significant Bit
NA4 Naturally aligned four-byte Region
NAPOT Naturally aligned power-of-two Region
OS Operating System
OTP One-Time Programmable
PCB Printed Circuit Board
PMP Physical Memory Protection
RAM Random Access Memory
ROP Return-oriented Programming
RTOS Real-Time Operating System
SBI Supervisor Binary Interface
S-Mode Supervisor-Mode

SoC System on Chip

TCB Task Control Block

TEE Trusted Execution Environment

TOR Top of Range

UART Universal Asynchronous Receiver/Transmitter

U-Mode User-Mode

1

Introduction

This chapter presents an overview of the thesis subject, highlighting its importance and the related research, followed by outlining the main contributions made by the thesis. Finally, the organization of the thesis subsequent chapters are presented.

1.1. Motivation

Embedded devices have experienced a huge increase in numbers, and they are finding their way into many areas of our lives. This can be in the form of intelligent sensors in our houses, smartwatches, headphones, medical implants, computing chips in cars, etc. The market for the embedded devices is steadily growing and forecasts [1] predict exponential growth for the next decade. As can be seen in Figure 1.1, the current market valued at over 500 billion USD, is projected to expand by a factor of six, surpassing 3 trillion USD in 2033. A recurring theme is that the embedded systems must be energy efficient to ensure a long battery life and remain affordable. This is also often true for Internet of Things (IoT) devices, which are embedded systems designed to connect and communicate over the internet. However, the security factor of these devices is often disregarded, as implementing effective security measures is normally expensive and time-consuming, and the pressure to reduce time-to-market exacerbates this issue. Furthermore, advanced attack techniques intensify this challenge by enabling attackers to physically expose embedded systems, resulting in invasive access to such devices and further complicating their secure design. Currently, the necessity of strong cybersecurity is globally recognized as a critical requirement for any internet-connected device. In fact, protecting against cyberattacks is becoming so important that the 2023 World Economic Forum put cybersecurity in the current and future top 10 risks globally [2]. They emphasize that as technology becomes increasingly central to our daily lives, it is crucial to address the associated risks.

This growing concern for cybersecurity is further complicated by the complexity of embedded systems themselves. These systems often integrate many software components with potentially different origin. Third-party software is frequently used, either from public online repositories or proprietary that they buy from external suppliers to reduce costs. With this increasing dependency on third-party software, vendors lose the direct control over the software they ship, which opens up many possibilities for attackers to infiltrate malware. This code might run alongside critical code that is responsible for the safe operation of the application. There are several examples where this did go wrong. One recent example of the vulnerabilities associated with third-party software is seen in the incident involving a system jointly managed by Microsoft and CrowdStrike [3]. This system experienced disruption during an outage that affected many airports and prominent corporations, mostly as a result of a faulty software upgrade by a third-party vendor. This event caused substantial interruptions in operations and revealed crucial security vulnerabilities in extensively depending on external software sources for vital functions. The consequences of such a failure emphasize the complex difficulties and even dangers of incorporating third-party components. Another significant example of the risks associated with software emerged in 2020, when security researches were able to upload malicious apps to Fitbit's official appstore [4]. Fitbit sells wearable devices such as smartwatches and fitness trackers. These devices

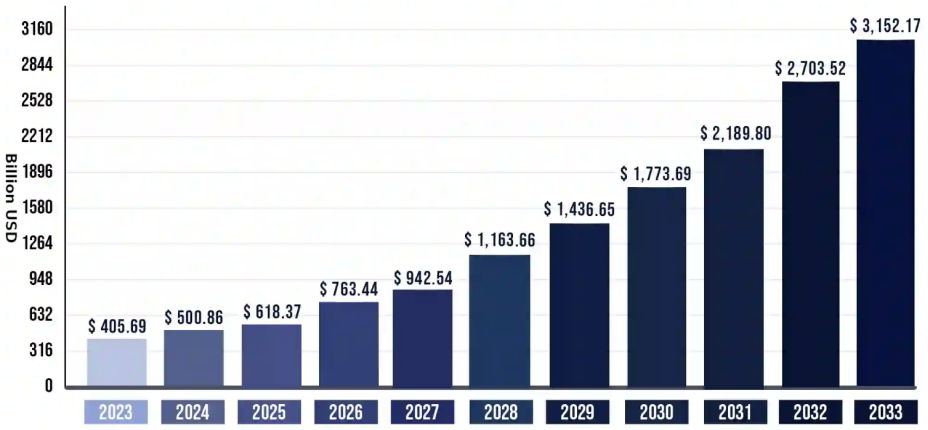


Figure 1.1: The global Internet of Things (IoT) market size in 2023 and prediction from 2024 to 2033 [1].

collect and analyze health and activity data like heart rate, steps taken, sleep quality, location, that are of personal and sensitive nature. Users can be tricked into installing the seemingly official apps that the researches managed to upload to the appstore. These apps could then be able to exfiltrate this sensitive data from the wearable devices and even the connected smartphones. This is a huge threat to the user's privacy because malicious actors could use this data for identity theft or personal profiling.

While privacy attacks already raise significant concerns, stakes are even higher if these attacks can put yours and others life at risk. In 2015, cybersecurity researchers were able to remotely hack into a Jeep Cherokee [5]. They demonstrated that they could exploit a vulnerability in the internet-connected navigation and entertainment system, which gave them access to critical vehicle functions, including steering transmission, and brakes while driving. Figure 1.2 shows the researches remotely disabling the car's breaks that caused it to crash into a ditch. Modern cars contain many of small embedded systems that control different vehicle functions and communicate with each other. This incident vividly demonstrates the danger of vehicles becoming more interconnected. While it enhances the functionality of cars, it also increases the attack surface, which makes robust security ever more important.

Figure 1.2: Charlie Miller and Chris Valasek carrying out their Jeep Hack attack [5].

The increase in number of embedded devices and there connectedness to each other and the internet introduces completely new possibilities for attacks, especially if the devices are not secured. One might think, that a small IoT device might not be a big danger, until hackers make these devices cooperate. The Mirai botnet [6], in 2016, was able to inject malware into 600,000 IoT devices that were easily hackable, i. e. devices use default passwords, or no security at all. These devices together were able

to launch large scale attacks to permanently take down servers of Netflix, Amazon and Twitter. This incident highlights the urgent need for robust security solutions for embedded devices.

To understand the presence and implications of these security threats, it is useful to grasp the technologies that are currently being used on the market. Especially the operating system and processors used, the primary components of any embedded system, play a significant role in the security of the device. Depending on the application, these systems can vary significantly on the operating system and processor used. For instance, there are multiple operating systems that are commonly utilized for embedded systems, as highlighted in the 2023 Embedded Markets Study [7]. The study offers insights into current trends, challenges, and needs in the field of embedded operating systems. It includes the experiences of 655 engineers working on embedded applications, and it has also been conducted over more than 20 years now, which makes it a reliable source for understanding and identifying relative trends in embedded devices. There are several interesting findings about the current situation in embedded development. Among the myriad findings, it leaps to the eye that most attention is devoted to real-time capability, as shown in Figure 1.3. The study also reveals which embedded OS are currently used or considered to be used in the next 12 months. FreeRTOS and Embedded Linux are most popular with each 44%. The third most popular embedded OS is Ubuntu with only 12%. The distinction between these operating systems is clear. FreeRTOS is a minimalistic real-time kernel, offering a direct implementation that is ideal for small, resource-constrained applications requiring precise timing and low overhead. This makes FreeRTOS particularly well-suited for simple embedded systems and real-time applications where efficiency and predictability are important, while Embedded Linux offers a much richer OS experience with optional real-time capabilities, which makes it more suitable for more complex applications such as multimedia processing. Ubuntu, on the other hand, is a full operating system for desktop PCs and Laptops like Windows or MacOS, and does not focus on real-time, although technically possible [8].

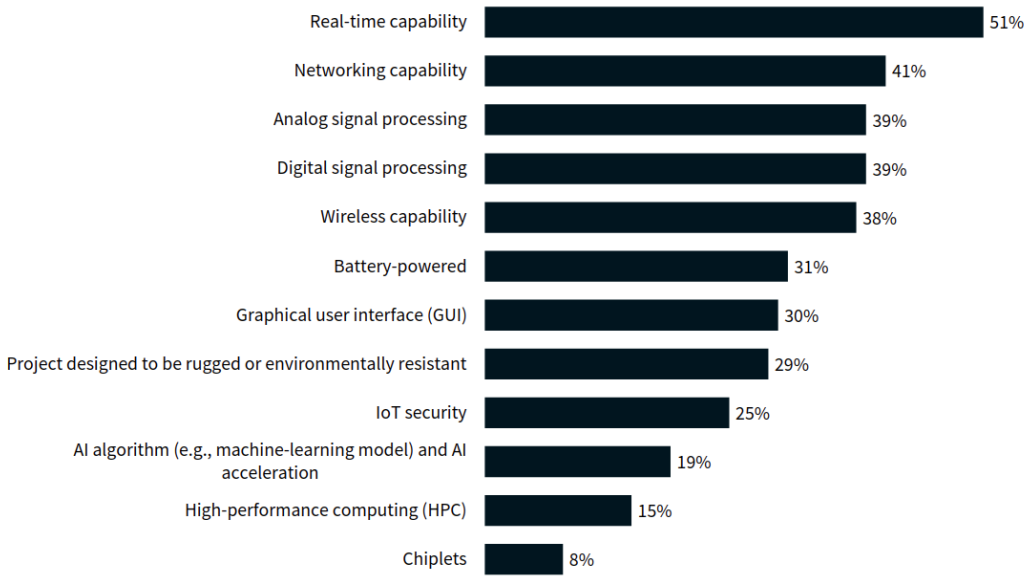


Figure 1.3: Answers on the question “Which of the following capabilities does your current embedded project include?” (multiple answers allowed) [7].

The efficiency of FreeRTOS stems from its lightweight kernel and the fact that all software components have direct control over both software and hardware resources, imposing minimal restrictions on performance. This works well and is reliable for well-tested and benign code. However, for hackers, this open-door in architecture presents significant security vulnerabilities to exploit. If robust security measures are not in place, it is not only simple for an attacker to gain access to the platform, but it also allows them to control the entire system behavior, potentially causing real harm to critical applications. It is not FreeRTOS that has suddenly become insecure. This RTOS has been in use for a considerate amount of time and reached a high level of maturity. Rather, it is the world around that has changed. The increase of the number of connected devices and the unprecedented reliance on third-party software

makes it necessary for FreeRTOS to adapt to these changes.

Besides the operating system, the choice of processor architecture is another critical factor. ARM and RISC-V are common examples of architectures frequently utilized in embedded systems. ARM is popular commercial Instruction Set Architecture (ISA) that is often used for the processors of smartphones and embedded devices because it is more lightweight aims at energy efficiency. ARM does not build processors themselves but they earn money with license fees that they get from manufacturers that want to use their ISA [9] [10]. RISC-V, on the other hand, is a general-purpose ISA developed by UC Berkeley and it is BSD licensed. The philosophy of RISC-V is to create a free ISA that is open to all users. The goal was to have a clean and universal ISA that can be used for many different use cases. It is highly extensible and this allows researches to build specialized computing systems. Not having to deal with licensing models and the growing tool support, speeds up research as well as the development of commercial applications. The European Processor Initiative sees the potential in this open ISA and funds the development and production of RISC-V based high-performance computers [11]. Market studies predict that there will be 60 billion RISC-V cores by 2025 [12].

As the use of FreeRTOS and RISC-V processors continues to increase, it becomes even more crucial to address the security challenges associated with them. While there are numerous studies on the security of FreeRTOS and embedded systems using ARM processors, the security analysis of FreeRTOS on RISC-V processors remains relatively underexplored. FreeRTOS is a relatively simple OS and focuses on being as lightweight as possible and keeping the kernel small. Focusing on energy-efficient designs, the logic in the CPU is often kept simple. While CPUs in desktop computers are equipped with hardware support, the Memory Management Unit (MMU), that completely isolates processes running on it, this is not viable for embedded applications because of the high area complexity that would result in high power consumption. As a consequence, for embedded applications it is very typical to have a flat memory model, i.e. all the code lies in the same memory space just next to each other. This reduces the complexity of the design significantly. This makes it also possible for tasks to communicate to each other but it could also be abused by malicious applications. When thinking of life critical situations as automotive or health related system, it is of utmost importance to prevent malicious read and write accesses to critical code because of their severe consequences. A third-party application could, for example, read sensitive health data or impair important safety features in a car.

In its standard version FreeRTOS does not implement any serious security protection. [13] demonstrates several Buffer Overflow (BOF) based attacks on FreeRTOS, which proves how vulnerable FreeRTOS based systems are without any mitigations. Therefore, it is crucial to develop secure solutions for FreeRTOS-based systems on RISC-V processors. This effort is essential to ensure robust security measures are in place, protecting these systems from potential threats and enhancing the overall reliability of embedded applications.

1.2. Related Work

The ubiquity of this this OS led to various attempts to make FreeRTOS more secure. One of these attempts are made by FreeRTOS itself as they release a more secure version of FreeRTOS that has Memory Protection Unit (MPU) support [14]. An MPU is a hardware unit inside a microprocessor or microcontroller that can be configured to protect memory regions from being accessed. When a task is running, the FreeRTOS kernel reconfigures the MPU so that the task is restricted to its regions. This addresses the security weakness of the flat memory model. The FreeRTOS changelog [15] reveals that the first official MPU version of FreeRTOS was actually already released back in 2009 for the ARM Cortex M3 family. However, for the most time, there were only minor updates and not many platforms were supported. There has not been much progress for a decade, but from 2020 onwards, major improvements have been made and the MPU version received significantly more attention. However, official MPU support still exclusively exists for a few Arm platforms [16].

For RISC-V, only very limited efforts have been made. SiFive published a basic secure RISC-V version for FreeRTOS in 2020 [17]. This port uses RISC-V's Physical Memory Protection (PMP) which is similar to an MPU. The implementation mainly only covers the integration of the PMP into the kernel. In essence, only the task's stack is protected and the executable code for the tasks are all put into some unprivileged code area without being protected from each other. A simple example implementation [18]

exists but is far from secure. Eventually, the repository has been archived and is not longer maintained. All in all, this implementation is incomplete but some code of it is being used in this thesis. Zephyr, an Real-Time Operating System (RTOS) also designed to run on resource-constrained devices, is some steps ahead and released an official PMP version for RISC-V in 2021 [19]. It would be interesting to know how the real-time performance is impacted by these additional security measures, however, to the best of my knowledge, no literature on the performance and area impact is available for the secure version of Zephyr. This raises the question: Why is embedded security, one of the most pressing problems of our time, being neglected for FreeRTOS, the most widely used RTOS, for RISC-V? Could it be that adding security to a lightweight RTOS comes with unacceptable performance degradation? Or is the implementation too complex for programmers to deliver competitive products?

Given the critical importance of security in embedded systems, this thesis investigates a variety of methods for improving the security of FreeRTOS on RISC-V System-on-Chip (SoC) architectures. This is done in light of the fact that there has been a limited amount of study conducted on the subject of FreeRTOS security on RISC-V, which is becoming an increasingly popular processor, and the vital relevance of security in embedded systems. The focus is on developing and evaluating security measures that address the unique challenges posed by the RISC-V architecture while ensuring that FreeRTOS, a widely used real-time operating system, remains robust and secure in embedded environments. This research aims to advance the state of the art by proposing innovative solutions that mitigate potential vulnerabilities and enhance the overall security framework of FreeRTOS on the RISC-V processor.

1.3. Contributions

This thesis is going to investigate the challenges and trade-offs involved in integrating security into FreeRTOS for a RISC-V platform. The implementation is inspired by the existing FreeRTOS-MPU version for ARM microcontrollers [14]. Further, the impact on the real-time performance is evaluated and compared to the conventional FreeRTOS implementation. Embedded security requires an holistic approach, for that reason attacks on typical embedded system are analyzed and a comprehensive strategy to secure the entire system is proposed. The detailed contributions made in this thesis are listed below:

1. **Developing a secure SoC based on the CV32E40S RISC-V core devised by OpenHW Group** The CV32E40S RISC-V is considered to be the state of the art implementation within the RISC-V architecture. The core implements the Physical Memory Protection (PMP) mechanism as in the latest RISC-V specification which plays a pivotal role in enhancing the security of the system. However, as stand alone, the RISC-V cannot provide a security solution. Therefore, it is crucial to integrate the core within a system that includes caches, memory, and peripherals to ensure a fully secure solution.
2. **Designing and implementing PMP support techniques for FreeRTOS that can prevent malicious software from accessing critical data** Isolating tasks from each other and from FreeRTOS kernel code is a way to drastically enhance the security of FreeRTOS. This thesis presents the design and implementation for this on a RISC-V platform that is equipped with PMP units.
3. **Adding PMP support for dynamic memory allocation in FreeRTOS** Building on top of the previous contribution, PMP support for dynamic memory allocation in FreeRTOS has been developed and implemented. Such a feature has not previously been implemented on either the RISC-V or the MPU version of FreeRTOS for ARM platforms.
4. **Implementation of a secure boot procedure** The FreeRTOS kernel makes sure that the tasks cannot misbehave but an attacker could modify the kernel to circumvent these measures. A secure boot has been implemented that detects whether the kernel has been modified. The procedure is inspired from the Keystone framework [20] and uses SHA-3 as a hash functions and Ed25519 for the signature verification.
5. **Integration and improving the security of the external memory encryption unit from previous thesis [21]** A lightweight encryption unit for the external memory has been integrated, using the Prince encryption scheme. Additionally, two major improvements have been done: First, selective task encryption connected to the PMP mechanism. Second, address-depended encryption to avoid determinism between memory cells.

6. **Investigation of attack scenarios for the embedded platform** This thesis offers an overview about the known attacks on embedded devices. Then, it focuses more on the software attacks and develops a threat model for a typical embedded system.
7. **Evaluation of the impact of PMP integration in FreeRTOS on real-time performance** The proposed design is implemented on a Xilinx PYNQ-Z1 FPGA platform to investigate the effect the PMP support for FreeRTOS has on the real-time performance. Elaborate test cases were run for three selected crucial real-time properties: Task switching time, mutex acquiring and releasing, and dynamic memory allocation time. Further, the area overhead is reported.

1.4. Outline of this Thesis

This thesis is organized into seven chapters. Chapter 1 outlines the motivation, the related work, and the contributions of this thesis. Chapter 2 provides the relevant background about embedded system architectures, the RISC-V instruction set architecture, memory architectures, FreeRTOS, and some useful information about the C toolchain. Chapter 3 gives an overview over embedded security in general, and explains attacks and countermeasures that are relevant for this thesis in more detail. Chapter 4 presents the state of the art for securing FreeRTOS on RISC-V platforms, and then goes on by deriving requirements for the secure FreeRTOS implementation presented in this thesis, which results in a proposed a high-level design as a reference for the following chapters. Chapter 5 describes the design and implementation of this work in detail. It begins with choosing a suitable RISC-V core, and then describing all the implemented features of the platform that enhance the security of FreeRTOS. Chapter 6 demonstrates the effectiveness of the implementation and evaluates the real-time performance impact of the PMP integration into FreeRTOS. Further, it measures the performance impact of the integrated encryption unit and reports the area overhead, and it concludes with a discussion over the findings. Chapter 7 includes a summary of this thesis and suggests future work.

2

FreeRTOS on RISC-V System

Embedded systems are specialized computer systems that carry out specific tasks, either as standalone device or within bigger systems. They play a crucial role in several applications, spanning from common consumer electronics to essential infrastructure in sectors including automotive, aerospace, healthcare, and telecommunications. The market for embedded systems is seeing tremendous growth on a global scale, and it is anticipated to surpass a market value of 173 billion dollars by the year 2032 [22]. The expansion has been accelerated by technological breakthroughs such as the Internet of Things (IoT) [23], artificial intelligence (AI) [23], and 5G [23]. These technologies are being rapidly integrated into embedded systems to improve their usefulness and connection. This chapter introduces the reader to embedded systems and provides the necessary background on the components that are the main focus of this thesis, including the RISC-V processor, memory design, FreeRTOS operating system, and toolchain software. It starts with an introduction to embedded systems in Section 2.1. Followed by a detailed explanation of the RISC-V Instruction Set Architecture in Section 2.2. Section 2.3 provides detailed information on memory design. Section 2.4 explains the FreeRTOS operating system, and finally, section 2.5 covers the details of the toolchain.

2.1. Introduction to Embedded Systems Architectures

According to the Cambridge Dictionary, *"an embedded system is a computer designed to do one particular job or set of jobs as part of a larger machine"*. These systems are widespread, serving as the basis for various applications, including consumer electronics like smartphones and smartwatches, as well as advanced essential infrastructure such as automotive systems, medical devices, and industrial control systems. Developing an embedded system requires a comprehensive understanding of its main elements and applications. These elements can be classified into hardware and software. We refer to the physical components of a system as its hardware elements. While software elements includes both an operating system and/or applications. It also includes toolchain which are a series of software programs that convert high-level code into machine code. We will delve deeper into each of these components in the following paragraphs.

2.1.1. Embedded System Hardware Components

The hardware part of an embedded system (see Figure 2.1), in its most basic configuration, has a **computational unit**, a **memory unit**, a **power source**, and at least one **input/output** component that allows the system to interact or communicate with its environment. Additionally, the system must be provided with a **clock** that drives the computation units and communication busses, and some kind of **reset logic** that puts the system into a defined initial state [24][25].

- **The Computation Unit** also known as the processor, is the heart of an embedded system, performing the majority of the system's processing tasks. It executes instructions from software programs, carrying out essential arithmetic, logic, and control operations. Essentially, the processor acts like the brain of the system, interpreting and processing data received from the memory unit, executing the necessary computations, and then sending the results back to the memory or

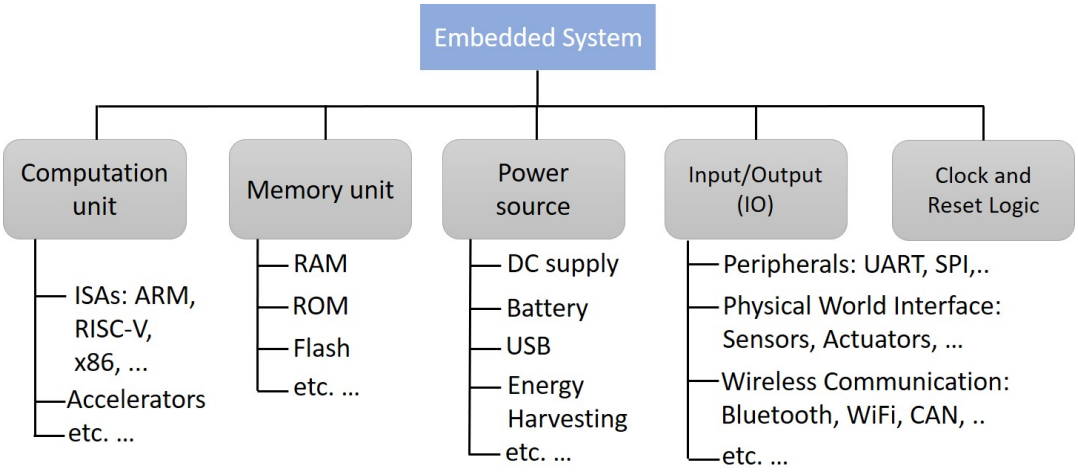


Figure 2.1: Embedded system hardware components.

other hardware components for further action. The Instruction Set Architecture (ISA) of a processor specifies how the instructions have to be interpreted. ARM and RISC-V are two examples of embedded system ISAs that are becoming increasingly popular. x86 is more a complex ISA that is often found in the processors of servers and laptops. Accelerators can be used to speed up applications-specific computations. For example, cryptographic operations, like encryption, could be an operation that a system has to perform frequently. Having a dedicated accelerator for that can take load from the CPU and execute the computations more efficiently.

- **The Memory Unit** is a type of data storage that stores the program code and other necessary data for the operation of the embedded application. It plays a crucial role in the performance, reliability, and efficiency of an embedded system. There are three types of memories that can be used in embedded systems:
  - RAM (Random Access Memory): A volatile memory type used for temporary data storage during operation. There are two different types of RAM used in embedded systems: SRAM (Static RAM) is faster and more reliable, as it does not need to be refreshed periodically. It is used for cache memory due to its speed. DRAM (Dynamic RAM): Unlike SRAM, DRAM needs to be refreshed thousands of times per second. Despite this, main memory uses DRAM because it's less expensive than SRAM. DDR (Double Data Rate) is a faster variation of DRAM that is capable of transferring data on both the rising and falling edges of the clock signal, effectively doubling the data transfer rate.
  - ROM (Read-Only Memory): A non-volatile memory that stores firmware and other essential software that does not change frequently. This guarantees the correct boot and operation of the system even after power off. Flash memory is another type of non-volatile memory that allows for data storage and retrieval, making it useful for firmware updates and data logging.
  - Flash memory merges the advantages of RAM and ROM, enabling data rewriting and data retention without power consumption. Cache: A smaller, faster type of volatile memory that provides high-speed data access to the processor, significantly improving processing speed by storing frequently accessed data and instructions closer to the processor.
- **The Power Source** for embedded systems can vary widely, including batteries, USB connections, and direct AC power supplies, depending on the application and portability requirements. For instance, portable devices often rely on battery power, while stationary systems may use AC power. Some low-power devices are even able to use the ambient energy available in its environment through energy harvesting. Examples could be implanted sensors for medical applications, or sensors that monitor the tire pressure in automobiles [26].
- **Input/Output Components** can be included in various forms, depending on the system's purpose. Peripherals allow the system to connect to other hardware using standard interfaces such

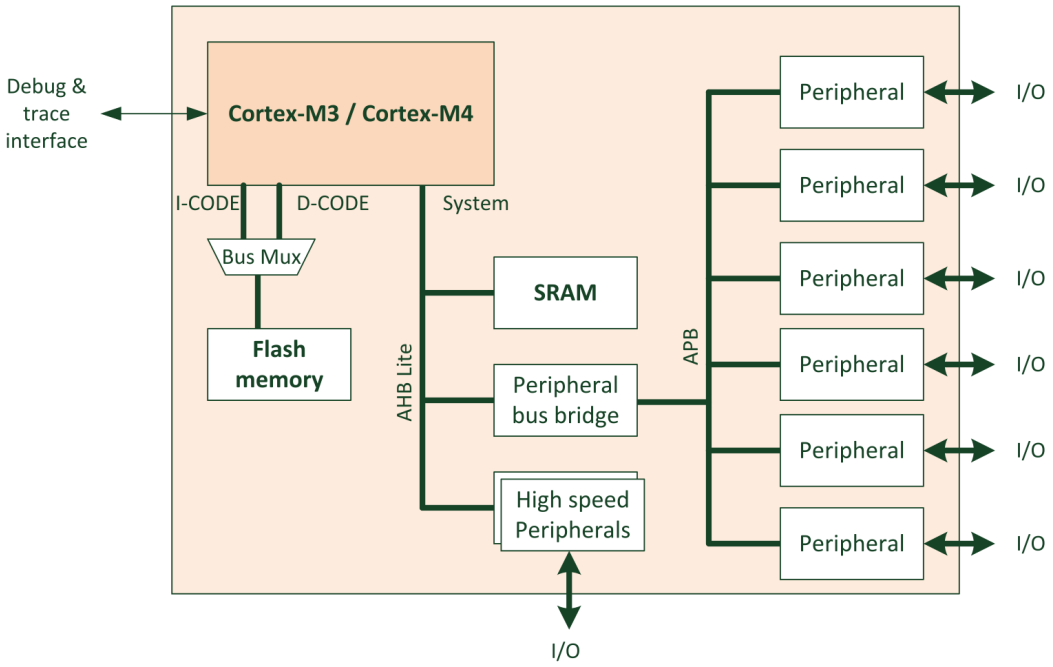


Figure 2.2: A simple Cortex-M3 or Cortex-M4 processor based system [28].

as UART, I2C, and SPI. Wireless communication components enable the system to communicate wirelessly with other systems or a cloud server. Sensors allow the system to interact with the physical world by collecting data like temperature, pressure, or motion.

- **The Clock and Reset Logic** can be regarded as the conductor of the whole system. The clock synchronizes all components within the system, ensuring that they can cooperate at the correct timing. The reset logic, on the other hand, is responsible for initializing the system to a known state upon power-up or when a reset condition occurs. This prevents that the system finds itself in an undefined state after start-up.

Figure 2.2 shows a typical embedded system. A suiting reference system is the ARM Cortex M3 microcontroller family, which is not a RISC-V platform, rather, it is one of the most mature and well established controller families in the embedded market. Multiple semiconductor manufacturers, such as NXP, Texas Instruments, and STMicroelectronics, implement the design in their products and provide extensive support. RISC-V is a relatively new ISA, thus there has been less time for established platforms to emerge. This could certainly change in the future, but for now, the ARM Cortex M3 family remains the leading representative for an embedded computing units due to its wide adoption and mature ecosystem. The central component of the system in the figure is the CPU core, here described as “Cortex-M3/Cortex-M4”. The core is connected to flash memory that it accesses via an I-CODE bus for instructions, and a D-CODE bus for data. A bus multiplexer manages the accesses to avoid access collisions. Another system bus connects the core to an additional SRAM memory unit and peripherals. Normally, peripherals are accessed by a separate bus that operates with a different speed. This allows for power consumption optimization. In this example, the Advanced High-Performance Bus (AHB) Lite is used when high speed operation is required, and the Advanced Peripheral Bus (APB) for lower bandwidth peripherals. These busses are part of the Advanced Microcontroller Bus Architecture (AMBA), which is an open-standard for on-chip interconnection of functional blocks in SoC designs. This open standard was developed by ARM and contains a different protocols for different requirements. An overview is given in Figure 2.3. At the bottom there is APB, being the simplest bus for low bandwidth peripherals. AHB can be used for more complex interconnects and operates at higher speed. Above that, more complex buses exist, such as the AXI, ACE and CHI. More details can be found in [27].

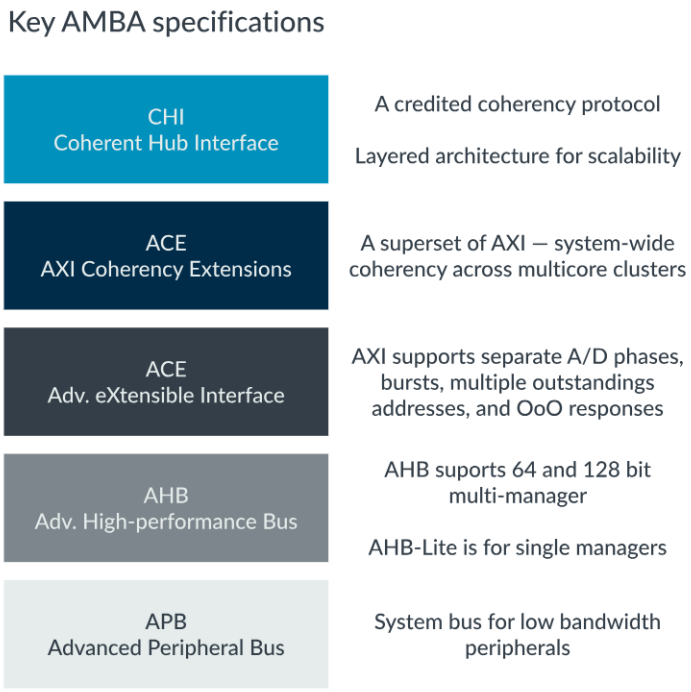


Figure 2.3: AMBA specifications ordered by performance [27].

2.1.2. Embedded System Software

The software part of embedded systems typically consists of toolchains, applications and/or operating systems. The toolchain is a critical element in the development process, providing the necessary infrastructure to convert high-level code into machine language that the processor can execute. Compilers, assemblers, and linkers are fundamental components of the toolchain, transforming high-level code into executable machine code. The compiler translates high-level programming languages like C and C++ into assembly language. This process involves syntax checking, code optimization, and generating assembly code. Popular compilers include GCC (GNU Compiler Collection) and LLVM. The assembler takes the assembly code generated by the compiler and converts it into machine code (binary instructions) that the processor can understand. This step involves translating mnemonic instructions into their corresponding binary format. The linker combines various pieces of machine code and resolves references between them to produce a single executable binary. It ensures that the final program has all the necessary components, including libraries and dependencies, correctly placed in memory. Applications in embedded systems are developed to execute particular functions such as data collection, signal processing, control tasks, and user interface management. These applications are highly optimized for efficiency and reliability, tailored to the specific requirements of the hardware and the intended use case. For instance, an application in a smart thermostat would manage temperature data, control heating and cooling elements, and interface with the user to set temperature preferences. Operating Systems (OSs) in embedded systems are responsible for managing the hardware resources, providing essential services to the applications, and ensuring that operations are performed efficiently and reliably. For the rest of this subsection, we will focus primarily on operating systems.

In many cases, established embedded OSs are chosen to facilitate the development of embedded applications. It is a recurrent theme that embedded systems have to execute a given set of operations regularly in a specified amount of time. For example, a system connected to a humidity sensor in a plant pot is used to monitor the humidity of the soil to know whether it is time to water the plant again. This measurement shall not only occur once but frequently, with a specific time period. There could also be other tasks that the system performs. There could be a motor that is watering the plant after the sensor task detects that the soil is too dry, or the temperature is measured completely independently for some other purpose. The programmer has to implement some kind of scheduler that decides when which task has to run. It also has to manage the access to system resources, and considers what

happens if two tasks want to write to the same file at the same time. This has to be taken care of, or one task could write to the file before another task has finished writing, causing the file to become corrupted. This is a very common use case for an embedded OS. Instead of first having to write all functionality for handling different tasks in a specified time window and handling all the possible edges cases, like what happens if two tasks are triggered at the same time; multiple tasks try to use the same hardware resources; different tasks have very different priorities because they are more critical, and so on, an off-the-shelf OS can be used to decrease the development time and many possible bugs can be avoided because smart developers and a broad community have already thought about this. There are different OSs that implement differently complex schedulers. Figure 2.4 presents a qualitative overview of OSs and there suitability depending on the processing power of the system it runs on.

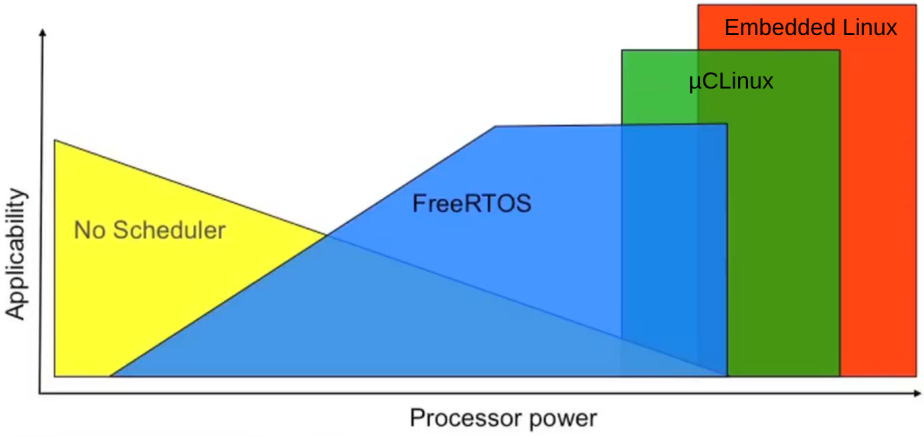


Figure 2.4: Applicability of different embedded operating systems based on the host system's processor power [29] (modified).

Bare-Metal

For very simple cases an OS is not needed and the program runs directly on the hardware. This is called *bare-metal* programming and is here represented by the yellow region saying "No Scheduler" because it uses a custom scheduler implementation. In this case, the programmer needs thorough knowledge of the platform because there is no level of abstraction, unlike when using an OS. Bare-metal programs can be highly efficient and optimized for a given application. However, when programs get more complex, the increasingly complicated resource management increases the development time and makes the implementation more error-prone [24].

FreeRTOS

FreeRTOS is designed for slightly more complex applications, providing a lightweight, real-time operating system that offers a minimal but complete level of abstraction. This abstraction simplifies resource management and task scheduling. In computing, "real-time" refers to systems or processes that respond to inputs or events within a guaranteed time frame. For embedded systems, this is often crucial because they often run applications where delays could lead to unacceptable outcomes. It is an often used Real-Time Operating System (RTOS) for low-performance embedded systems that is maintained under the stewardship of Amazon Web Services. The FreeRTOS kernel and the associated libraries are distributed for free under the MIT open source license. It focuses on microcontrollers and small processors by making it scalable size, with usable program memory footprint as low as 9KB. Some architectures include a tick-less power saving mode. It is also highly portable by supporting more than 40 Microcontroller Unit (MCU) architectures and more than 15 toolchains. This is achieved by having a fixed kernel that is used for every architecture and a porting layer that bridges the gap between kernel and architecture specific hardware [30]. Because it is the main software component used in this thesis, it is introduced here only briefly and described in detail in the following in its own dedicated section.

Embedded Linux

A popular operating system that experiences widespread use is GNU/Linux, or commonly referred to simply as "Linux". Linux is a Unix-like OS, first released by Linus Torvalds in 1991. Unix is the predecessor OS the development of which began in 1969 by Ken Thompson, Dennis Ritchie, and

Douglas McIlray at AT&T Bell Laboratories. What made Unix so successful was that it was portable, meaning it was not strictly bound to a specific computing platform. Before, operating systems were bespoke pieces that were written in assembly language for the target platform. Unix was written in the high-level language C that could be compiled to several platforms. Unix has always been a commercial product and is not freely available to the public. Numerous Unix derivatives have emerged, each with distinct licenses, leading to complex and cumbersome commercial use and licensing models. Two different developments, that had the vision of making software free to the public, led to Linux how we know it today. First, Richard Stallman initiated the GNU (GNU is not Unix) project as an attempt of creating a free Unix-like OS. The development on the kernel, the core part of the operating system, was too complex and the faced numerous delays. The second development was the Linux kernel developed by Linus Torvalds. The Linux kernel was inspired by a previous project of Andrew S. Tanenbaum called Minux, which could be licensed for educational purposes only. Frustrated by this restrictive licensing model, Torwalds began the development of his own kernel which resulted in the Linux kernel. The rapid development and stable performance made the developers of the GNU project stop the development of their kernel and integrated Linux instead. This is how the GNU/Linux OS and today it as disseminated as *distibutions*, like Ubuntu, Debian, openSUSE, etc. [31]

GNU/Linux was developed for systems like PCs and Laptops, as well as servers. These systems most often are not resource constraint as embedded systems that we are concerned with in this thesis. The Linux kernel is a fairly big and complicated piece of software, and running it on embedded devices would be very inefficient. However, GNU/Linux is open source and easily customizable to make it suitable for embedded applications. These customizations are referred to as *Embedded Linux*. Advantages of this are that the kernel is fairly mature and stable, no licensing fees, and ease of use for developers who are already familiar with Linux. Still, the complexity of the kernel and the fact that it is not inherently developed for embedded systems make it less efficient in performance and memory footprint than other embedded OSs. Moreover, Linux lacks real-time support, which is often needed in embedded applications, as discussed earlier but this can be added by real-time patches [8] to the kernel.

Another disadvantage is that it is designed to work with an Memory Management Unit (MMU), and therefore, needs three privilege levels. In RISC-V these are the M-mode for the bootloader and firmware, the S-mode for the Linux OS, and the U-mode for the processes [32]. As explained before, embedded systems normally do not implement an MMU because it is too resource intensive. Figure 2.5 shows how Linux uses these different privileges when run on a RISC-V platform. The start-up code in the ROM code and the 1st stage bootloader need highest privileges to configure the hardware in M-mode. The there is the Supervisor Binary Interface (SBI) that is the provides M-mode functionality to the Linux kernel. In this case OpenSBI is used which is a open source SBI for RISC-V. Having an SBI makes a clear separation of privileges possible. The complex kernel should not directly run in M-mode for security reasons, but rather in S-mode. Then there is a second U-Boot bootloader that runs in S-mode that sets up and starts the Linux kernel. The user applications run on top of the Linux kernel and are degraded to U-mode. μLinux is a fork of the Linux kernel that addresses the problem that embedded systems typically do not contain a MMU but compared to OSs that are specifically designed for these systems it is still large and complex and worse in real-time performance [33].

The next part of this chapter will go into more detail about the RISC-V ISA, which impacts the computing unit, the memory architecture, the FreeRTOS operating system, and the toolchain that goes with. This is because the thesis is mostly about these parts and how they work together in terms of embedded systems security.

2.2. RISC-V Instruction Set Architecture

If a programmer writes a program in some programming language, it has to be translated into something that the Central Processing Unit (CPU) of the system can understand. For the C language, the compiler takes the human-readable C source code and generates machine code from it. Eventually, this machine code consists of zeros and ones that the CPU interprets as instructions. Which instructions there are, and which series of zeros and ones corresponds to what instruction, is defined by the ISA, as well as the expected behavior of an instruction when executed by the CPU. A CPU has to implement this behavior in hardware and can only correctly execute machine code that was compiled for the specific ISA of the CPU.

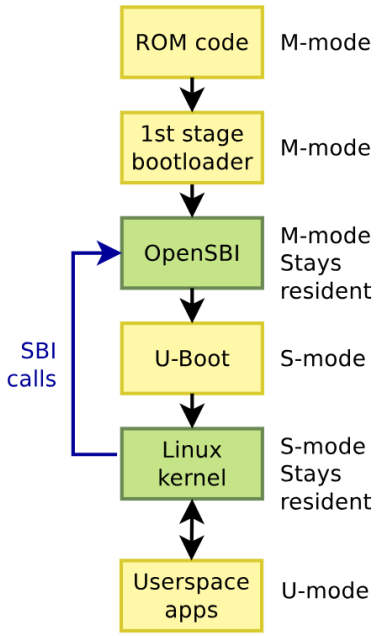


Figure 2.5: Interaction between different software components and their privilege levels in GNU/Linux on RISC-V [32].

RISC-V is an ISA with the initial intention of supporting research on computer architecture and for educational purposes, thus it is freely available for use for everyone. In the beginning, it was developed by researchers from the University of California at Berkeley and is now maintained by RISC-V International, which is a non-profit organization. RISC, which stands for *Reduced Instruction Set Computer*, uses a small, highly optimized set of instructions that are simple for the CPU to execute. On the contrary, CISC (Complex Instruction Set Computer) entails more complex instructions that accomplish tasks with less instructions. This requires more hardware logic and instructions can also take multiple cycles to be executed. This reveals that this RISC-V is designed with a focus on simplicity, aiming at efficient performance and potentially simplified hardware design. It also aims to be flexible and not overly tailored to a specific style of microarchitecture or implementation technology, such as Application Specific Integrated Circuit (ASIC) or Field Programmable Gate Array (FPGA). The “V” in its name was chosen because it is UC Berkeley’s fifth major RISC ISA design. RISC-V is fully described in two volumes of the ISA specification, [34] and [35].

Eventually, RISC-V successfully found its way into the commercial market [36], as its open design and flexibility constitute a valuable alternative for companies that want more control and freedom in their hardware design. The facilitated extensibility is especially well-suited for project-specific requirements. The biggest competitors among ISAs are x86 and ARM. Both of them are not available openly: x86 is completely closed source CISC ISA developed by Intel, while ARM is open but when used commercially, license fees must be paid to ARM Holdings which develops the Intellectual Property (IP).

It provides a stable base ISA and optional extensions. The base is the *integer* ISA, which is denoted by the extension name “I” and is prefixed by RV32 or RV64 depending on the register width. Among the optional extensions are the multiplication/division extension “M”, atomic instruction extension “A”, “F” or “D” for single- or double-precision floating-point extension, or “C” for compressed instructions, to name a few. Compressed instructions are shorter versions of standard instructions that reduce the size of code. This is important when compiling a program to a specific platform. The compiler must be configured to, for example, RV64-IMC to make sure that it uses valid instructions. On top of that, the ISA allows for custom non-standard extensions.

2.2.1. Privilege Architecture

RISC-V supports three privilege levels of execution. At any point in time, a RISC-V hardware thread (*hart*) is running at one of these modes. The Machine-Mode (M-Mode) is the highest privilege and is the

only mandatory privilege level for a platform. It has low-level access to the machine mode implementation and is inherently trusted. In contrast, the User-Mode (U-Mode) has a restricted access to the core's internals and is suitable for conventional applications that cannot be trusted. For example, processes running in U-Mode are not allowed to access the `mstatus` register. This register controls critical settings, for example, whether interrupts are enabled. In between these levels is the Supervisor-Mode (S-Mode) where normally the operating system normally resides. Table 2.1 shows the possible combinations of levels for different systems that trade off higher isolation for higher implementation costs.

Table 2.1: Supported combinations of privilege modes [35].

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems

2.2.2. Physical Memory Protection

RISC-V offers an optional Physical Memory Protection (PMP) feature that can be used to restrict read, write, or execute accesses to certain memory regions. This can be used to limit the physical memory access privileges of chosen parts of the software to make the system potentially more secure. Maximal 64 regions can be configured and the access rights read, write, and execute can be specified separately for each region. PMP is defined in the second volume of the RISC-V specification [35]. The PMP-checks are applied to all memory accesses during S or U-mode. They occur during instruction fetches and data accesses. It is also possible to restrict data accesses even for M-mode and, moreover, the CSRs for configuring PMP itself can be protected from M-mode. In the latter case, the configuration is locked, as the CSRs can only be accessed and altered in M-mode. They can only be accessed again after a CPU reset.

The regions and the privileges are configured in the Control and Status Registers (CSR) `pmpaddrx` and `pmpcfgx`, respectively. The `x` must be replaced by the index of the region (0-64). The figures Figure 2.6 and Figure 2.8 describe these registers as in the RISC-V ISA specification [34]. The definitions shown here are the 32-bit versions. For 64-bit they are structured differently as each register holds double the width.

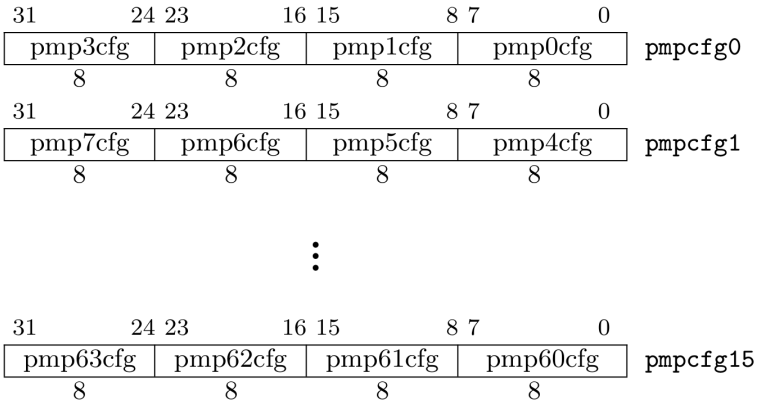


Figure 2.6: PMP configuration control and status register layout [35].

A PMP-`config` only needs 8 bit to be fully described. For that reason, one 32-bit register holds 4 configurations. The lowest three [2:0] bits hold the RWX (read, write, execute) permissions, the next lowest two [4:3] the addressing mode and the highest [7] holds the locked bit, as shown in Figure 2.7. Normally, the PMP settings only apply if the system is in the U-Mode. The locked bit can enforce PMP-rules also during M-mode but this is not relevant for this thesis. The address matching methods are more important and are explained in more detail.

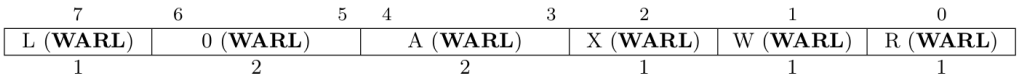


Figure 2.7: PMP configuration register format [35].

There are three different ways to describe an address range: Top of Range (TOR), Naturally aligned four-byte Region (NA4), or Naturally aligned power-of-two Region (NAPOT). Which of these address-matching methods is used is configured in the region's `pmpcfg` CSR. TOR specifies ranges by indicating the upper boundary of the region in `pmpaddrx`. The lower boundary is defined by the address of the previous region, or 0 in case it is the first region. NA4 specifies address ranges that are aligned to four-byte boundaries. With this, only 4 bytes or 32 bits can be protected by one region. NAPOT extends this concept to address ranges that align with any power-of-two boundary, offering flexibility in defining ranges that can vary significantly in size. The two bits in the field `A` of the `pmpcfg` register denote which addressing mode is chosen. Which value `pmpaddr` must hold is explained in Table 2.2. Note that NAPOT with a 4-byte range is effectively NA4 encoding.

Table 2.2: NAPOT range encoding in PMP address and configuration registers [35].

`pmpaddr`	`pmpcfg.A`	Match type and size
yyyy...yyyy	NA4	4-byte NAPOT range
yyyy...yyy0	NAPOT	8-byte NAPOT range
yyyy...yy01	NAPOT	16-byte NAPOT range
yyyy...y011	NAPOT	32-byte NAPOT range
...	...	...
yy01...1111	NAPOT	$2^{XLEN}-$byte NAPOT range
y011...1111	NAPOT	$2^{XLEN+1}-$byte NAPOT range
0111...1111	NAPOT	$2^{XLEN+2}-$byte NAPOT range
1111...1111	NAPOT	$2^{XLEN+3}-$byte NAPOT range

The register for the PMP-addresses is shown in Figure 2.8. Unlike the configurations, each register hold the address for exactly one region. The lowest two bits are not used because the minimal granularity is 4-bits, which means that they would always hold the value 0.

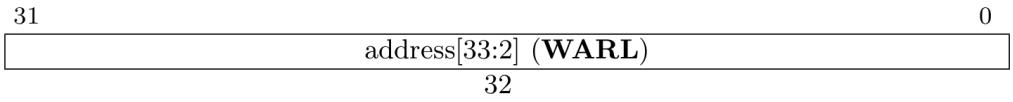


Figure 2.8: PMP address register format [35].

Failed accesses due to PMP violations generate an instruction, load, or store access-fault exception. Multiple exceptions are possible but which of them are eventually triggered and how they must be handled is not specified and therefore left to be decided by the programmer and developers.

2.3. Memory Architecture

When it comes to the design of memory for an embedded system, there are three essential aspects that must be taken into consideration: memory mapping input/output (IO), memory hierarchy, and memory segmentation. These components play a significant part in ensuring that the memory design is efficient, dependable, and secure. Memory mapping IO involves the assignment of specific memory addresses to various hardware components. By this technique, the processor is able to interface with peripheral devices by utilizing ordinary memory read and write operations. This simplifies the interaction between the devices and reduces the need for specific I/O instructions. The memory hierarchy is designed to enhance the tradeoff between speed and capacity in order to maximize both performance and efficiency.

Multiple levels of cache memory, main memory (RAM), and secondary storage are often included in the hierarchy. Memory segmentation is the process of splitting memory into distinct portions, each of which serves a distinct function, with the goal of improving both organization and security. The following subsections will explain in detail each of these aspects.

2.3.1. Memory Mapping IO

Embedded systems normally use special input/output peripherals to interact with the environment. This can be an on-board LED, hardware timers and counters, a serial communication interface like Universal Asynchronous Receiver/Transmitter (UART), network interfaces etc. The CPU accesses these peripherals to provide them output data, or to receive input data from them. One way to do it is by using a memory map that assigned addresses to various functions, memory types, or peripherals. Figure 2.9 is an excerpt of the pre-defined memory map for Cortex-M3 and M4 devices.

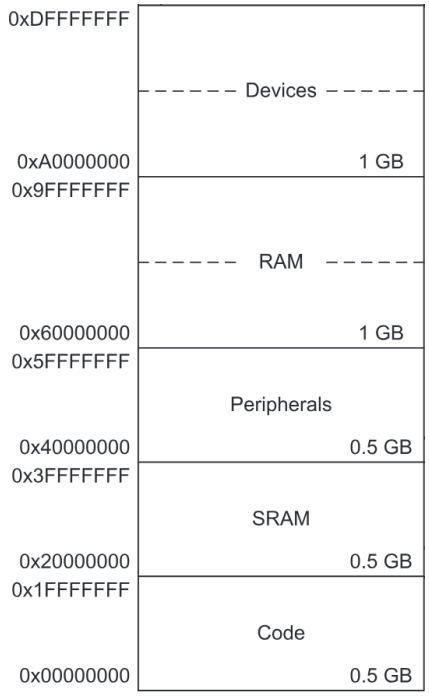


Figure 2.9: Memory map (excerpt) of Cortex-M3 and Cortex-M4 processors [28].

The lowest region of the map is the code-region, or the program memory. The processor expects the executable program code and the vector table to be located in this region, followed by another memory region that specifically uses SRAM as a memory type. The processor would access these two regions like normal memory. The third region is no memory region but a peripheral region, the region the processor can access the peripherals through.

For example, there could be a keyboard connected to the processor via some interface. The keyboard could hold two registers: KBSR (Keyboard Status Register) and KBDR (Keyboard Data Register). If KBSR is not zero, it means that the user pressed a key on the keyboard. KBDR would then hold the ASCII-character value of that key. KBSR could be mapped to the address 0x40000000, KBSR to 0x40000100. The processor can access the keyboard registers by reading from these memory addresses as if it was memory. To make this working, it requires additional control logic in hardware to connect the peripherals to the addresses. Figure 2.10 shows a simplified example datapath, the circuits and paths in the hardware, for the memory-mapped read logic. The processor is not shown but would be connected to the bus illustration as the bold arrow line at the top. The processor provides the address that it wants to read in the memory address register (MAR), and it expects to receive the value in the memory data register (MDR). Central for the memory-mapping mechanism is the address control logic, which is aware of the mapping table and controls the multiplexer to select the data from either the memory or the keyboard. If the address is 0x40000000, the KBSR value is selected and put

forward to the MDR. Likewise, if the address is 0x40000100, the KBDR value is selected, and if the address points at a memory location, it selects the value of the memory unit.

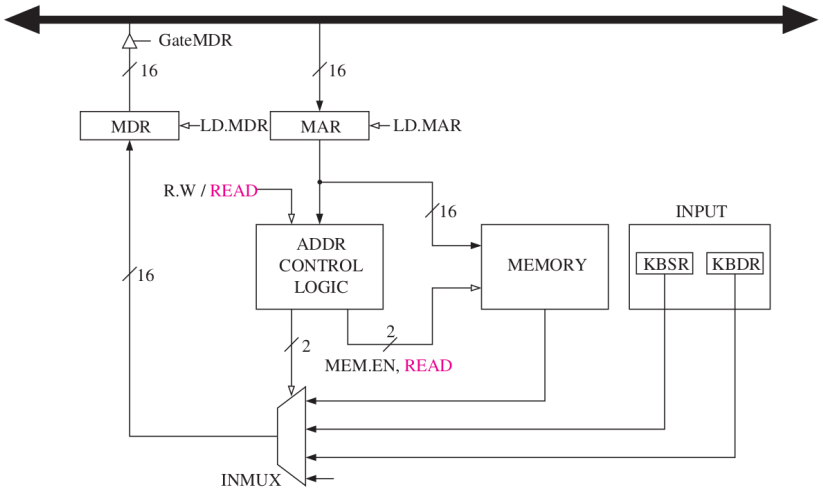


Figure 2.10: Example datapath for memory-mapped input of a keyboard. The keyboard is considered the peripheral in this case and holds the memory-mapped registers KBSR (Keyboard Status Register) and KBDR (Keyboard Data Register) [25].

2.3.2. Memory Hierarchy

In modern computing systems, memory accesses are usually relatively slow, compared to the processing speed of the processor [37]. The CPU can access the internal registers in one cycle but if it needs some data from the main memory or external flash, it needs to wait for many cycles until the data is requested and provided. The faster a memory is to access, the less data it can hold because faster memory is more expensive per byte. To optimize this cost-speed ratio, *memory hierarchies* are implemented, where systems integrate different types of memory with varying speeds and capacities. Figure 2.11 shows an example memory hierarchy and compares the different memory elements in a system by ordering them from fast (left) to slow (right). On the left, there are the internal registers of the CPU, which typically contain only a few bytes, and on the right side there are the main memory storages that hold the program code and data. To bridge the gap between fast and slow storages, caches are inserted that act as an intermediate storage layer that hold a smaller part of the memory but can deliver that data much faster. However, as the size of the caches are smaller than the actual memory, it is possible that the cache does not hold the data of a requested address by the CPU. In this case, the cache needs to request the data from the slower memory itself. This is called a *cache miss* and in this case, the cache does not achieve any performance gain. Therefore, hardware engineers try to design a cache that maximizes the number of cache *hits*, which means that the requested data is available and can lower the access time.

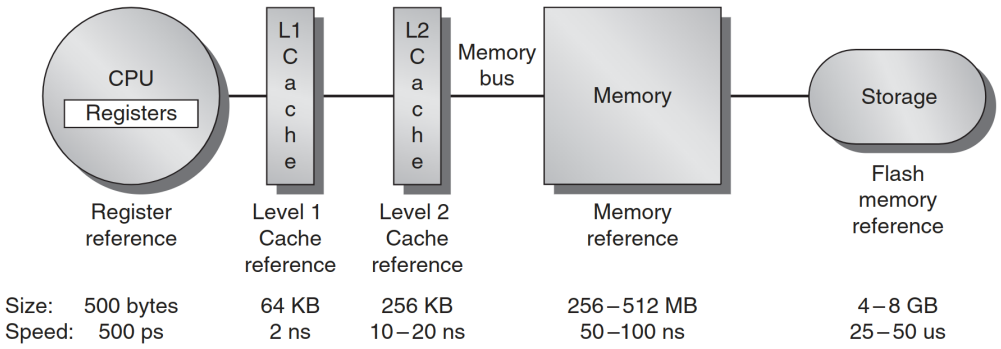


Figure 2.11: Memory hierarchy for a typical mobile device [37].

In the previous figure, two caches, level 1 and level 2, are being used but there can also be only one level or even more, depending on the requirements. Figure 2.12 shows the concrete memory hierarchy of an STM32F2/4 microcontroller from ST Microelectronics. There are two different caches for instruction fetches and data memory accesses that eventually access one unified flash memory unit [28]. An arbitration logic manages the accesses from the both caches to prevent read and write collisions. ARM does not define the memory system so they can differ between systems.

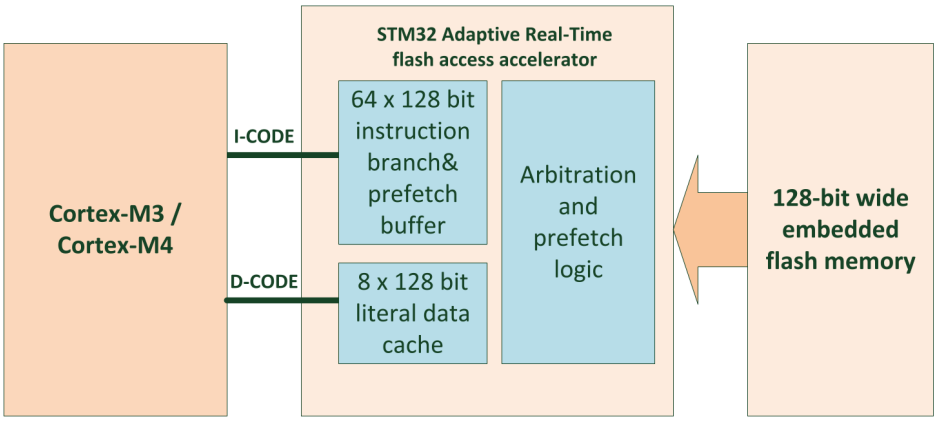


Figure 2.12: Concept of flash access accelerator in STM32F2 and STM32F4 from ST Microelectronics [28].

Caches should make use of two empirical principles: The principles of *spatial* and *temporal locality*. Most programs do not access code and data uniformly. The principle of spatial locality states that programs tend to access code and data from addresses that are close to recently accessed memory cells. The temporal locality states that memory that has been accessed is likely to be accessed again in the near future. Caches make use of these principals by loading not only the data of the requested address in case of a cache miss, but also adjacent data around this cell, called a *cache line*. Then, they will keep the data until it is replaced by more recent cache misses.

But what happens in case the CPU writes to an address? Is it only written to the cache or also to the main memory? There are two main methods. A *write-through* cache writes the to the cache and also directly to the main memory. Conversely, a *write-back* cache only updates the cache entry and writes to the main memory when necessary, i. e. when the cache line holding the data gets replaced. The former is simpler and ensures consistency between cache and memory. That is particularly useful in multi-core systems where each core its own cache. The latter is more efficient because it reduces the number of write operations to the main memory. However, it requires more complexity because the cache has to keep track whether a cache line has been modified.

Apart from that, it is possible to configure caches in many ways, including size of the cache in total, the length of a cache line, associativity, and replacement policy, to optimize system performance based on the application requirements. In general, a larger cache is likely to increase the hit rate, simply because it holds more data. Increasing the cache line can improve the hit rate by making more use of spatial locality, but it may also increase the misses if the extra data contained is not used often. Increasing the associativity means that data of an address can be put into more different locations in the cache. This added flexibility can be beneficial for the hit rate, however it adds some hardware complexity which can lead to higher cache access times. All in all, designers have to weigh all the trade-offs and decide on a suitable configuration for the use case.

2.3.3. Memory Segmentation

For the perspective of a running program, memory can be managed in different ways. The simplest form is the flat memory model that is often found in small embedded systems. The flat memory model offers no security and will be the main challenge for this thesis. With increasing complexity, the security can be improved by adding hardware units like the MPU or the MMU. For the work of this thesis, the MMU is not used but it is good to explain it here to give the full picture of possibilities.

Flat Memory Model

The simplest memory model is the flat memory model where a program running on the system has access to the whole memory space. The programs address data by their physical address. This is often the used model for small embedded systems because it does not add any hardware overhead to the processor. Nothing is stopping a program to access memory that is assigned to another program. This can be desirable to make efficient communication between programs possible. But it can also be undesirable when either a program accidentally misbehaves and corrupts data, or even when it acts maliciously. This can work well for simple applications with well-tested and trusted code, but with increasing complexity and integration of third-party code, additional protection might be wanted [38].

Memory Protection Unit

A Memory Protection Unit (MPU) is a programmable block inside the processor that enforces memory access permissions. When it is enabled it typically holds address ranges and the access permissions (execute, read, and/or write) to these address range. When a program violates the current MPU-configuration, the processor will detect this fault and handle it in some way. A program still accesses memory by using the physical address, the difference now is that the CPU intervenes if the access is not allowed as described by the MPU configuration. A MPU adds some hardware overhead but there are some use cases:

Security Management:

- Untrusted software components, can be restricted by the MPU. This also requires that the MPU is only configurable by a component of higher privilege, otherwise the untrusted component could deactivate it.
- Communication buffers for external communication interfaces can be secured by a MPU to prevent injection of malicious code. The data can be configured as non-executable memory and buffer overflows or buffer over-read (as in the infamous Heartbleed bug [39]) can be prevented.

Reliability:

- Trusted program can misbehave when not developed carefully. Pointers might accidentally point into other memory spaces or memory leak can consume more memory than planned, resulting in stack overflows. Both can be detected and prevented by a MPU.
- When high functional safety features are required, the MPU can ensure that software components cannot affect each other.

This and additional information about ARM Cortex-M3/4 processors can be found in [28].

Memory Management Unit

An Memory Management Unit (MMU) is normally implemented in all CPUs that run some kind of Unix-like operating system. In these operating systems there is usually a clear distinction between the *kernel space* of the operating system and *user space* where processes run.

This separation is achieved by an MMU which mainly does two things: It virtualizes a memory space for each user process, i. e. it divides physical memory into blocks and allocates them to different processes. It translates virtual addresses into physical addresses, i. e. each user process uses virtual addresses for accessing memory in a way that it appears to the process as if it has its own private, contiguous memory space. The MMU translate the addresses of the memory accesses by the process to the real physical address in memory, as illustrated in Figure 2.13.

Thus, an MMU makes it impossible for a process to access the memory that does not belong to it because the MMU translates all addresses of a process to one of its physical memory locations. However, this unit adds significant complexity to a CPU that results in a lot of additional area. Embedded systems usually do not incorporate this unit because it is important to keep the area and power consumption small [40].

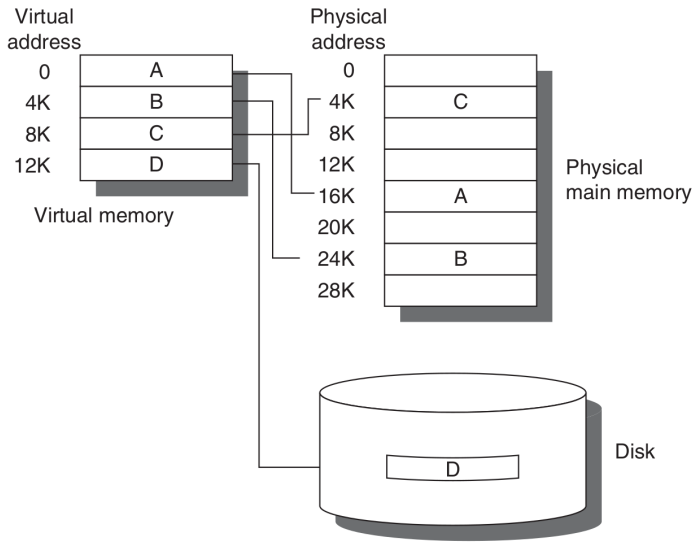


Figure 2.13: The logical program in its contiguous virtual address space is shown on the left. It consists of four pages, A, B, C, and D. The actual location of three of the blocks is in physical main memory and the other is located on the disk [38].

2.4. FreeRTOS in Detail

FreeRTOS, maintained by Amazon Web Services, is a widely used open-source RTOS for low performance embedded systems, distributed under the MIT license. As explained before, FreeRTOS is a lightweight, real-time operating system tailored for slightly more complex applications, which offers a crucial level of abstraction that simplifies resource management, as well as task scheduling, making it ideal for embedded real-time applications.

FreeRTOS offers several FreeRTOS ports various platforms. A port is a layer that connects the ISA and the hardware units to the FreeRTOS kernel. The kernel cannot operate independently and relies on peripheral units from the platform it is deployed in. For example, one of the most central signals for the FreeRTOS kernel is the “SysTick”, which acts as a system timer to generate interrupt requests at fixed time intervals. This mechanism is mandatory for the kernel’s scheduler to perform context switching between tasks. However, the way interrupts are handled on different platforms can differ significantly; also setting up the timer unit that interrupts the CPU is platform specific, thus, must be handled individually. This is where FreeRTOS ports are needed as they deal with these hardware-specific matters. Among many, FreeRTOS also provides an official FreeRTOS port for RISC-V, available for GCC or the IAR compiler [41]. In the following, the scheduler and the task management are explained in more detail.

Scheduler

Different RTOSs can have different levels of complexity, serving many different use-cases, each with unique restraints and requirements. The main component that ensures the real-time responsiveness of the RTOS is the scheduler, which manages the execution of tasks by prioritizing and allocating CPU time to them based on their urgency and importance. The classification of [42] provides useful context to understand how FreeRTOS relates to other RTOSs. FreeRTOS’s scheduling algorithm can be characterized as follows:

- Preemptive: A running task can be interrupted at any time to assign the processor to another active task, according to a predefined scheduling policy. On the contrary, non-preemptive algorithms run the running task until completion before another task can run.
- Static: The kernel’s scheduling decisions are based on fixed parameters, assigned to the tasks before activation. On the contrary, dynamic algorithms base their decisions on dynamic parameters that may change during the system’s runtime.
- Online: Scheduling decisions are taken during runtime. On the contrary in offline algorithms, the whole schedule is calculated before runtime and later executed by a dispatcher. As embedded

applications typically have to respond to more or less non-determined external interrupts, calculating the whole schedule beforehand is not possible.

- Heuristic: FreeRTOS does not use an optimal scheduling algorithm as it does not seek to minimize or optimize a global system cost function. The scheduler selects the task with the highest-priority that is ready to run, which is a heuristic approach as it simplifies decision-making but may not lead to the most optimal scheduling decisions.

FreeRTOS implements a lightweight, priority-based real-time scheduler. It makes sure that higher priority tasks, that are ready, are executed before lower priority ones. Tasks can be in one of five states and only one task can run at a time, see Figure 2.14.

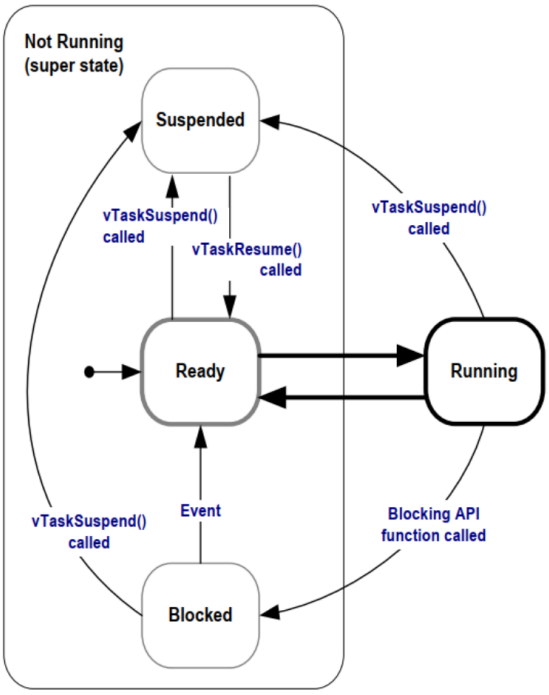


Figure 2.14: State machine for tasks in FreeRTOS [30].

There are two super-states: *running* and *not running*. When a task is running, FreeRTOS provides CPU-runtime to this task. It runs as long as the kernel stops it to run a different task, or the task calls an FreeRTOS Application Programming Interface (API) call that puts the task into the blocked state. API calls could be `vTaskDelay` and `vTaskDelayUntil` that a task requests to be delayed for some time. The first call blocks the task for a number of SysTicks, and the second call blocks the task for a number of SysTicks *relative to the previous activation time*, helping maintain a steady rate of execution. The FreeRTOS kernel manages this internally and activates the task again after enough SysTicks passed.

A task can also be *suspended* which can be thought of as disabling the task. Putting a task in this state usually needs explicit intervention from the scheduler. In normal operation this is not used. Rather, it is used to temporarily disable features. A task can be suspended also when being in any of the other states, so it is not further explained in the following.

When a task is in the blocked state, it means that the task is inactive and not ready to run yet. In the blocked state, the task is waiting for an event to happen. Either a timer delay is finished, or an external interrupt activates the task. In these cases it is moved to the ready state.

Tasks in the ready state are waiting to be executed but do not run yet because another higher-priority task is running. When the current task is finished, the FreeRTOS scheduler picks one of the ready tasks and runs it.

An example application with three tasks is shown in Figure 2.15. A fourth task, the idle tasks, is necessary because there must always be one task in the running state. For applications, in which it is

possible that none of the tasks is running, an idle tasks must be added that effectively does nothing and runs in lowest priority to make sure that any other task will preempt (pause it and runs instead) it.

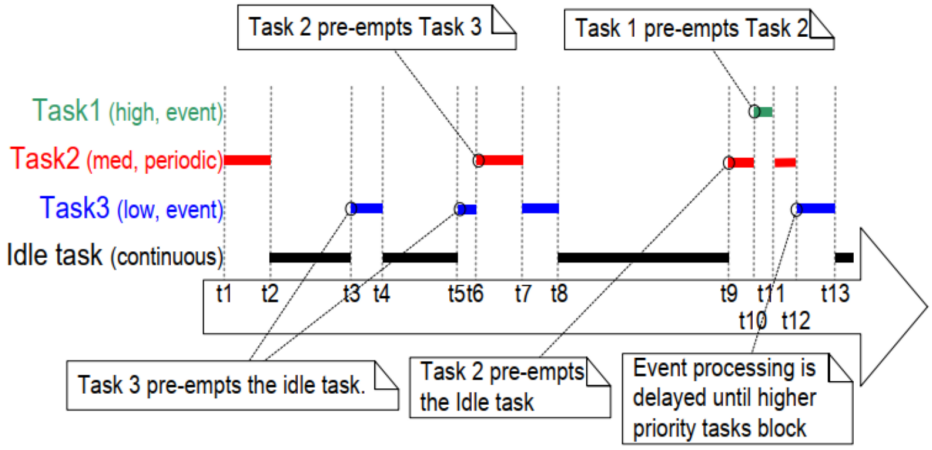


Figure 2.15: Execution sequence for three FreeRTOS tasks with different priorities [30].

In this example, there are three tasks with decreasing priority. The first and the third are event-driven and the second tasks is periodic. At t1, Task2 starts to run and finishes at t2 because no other task was ready during that time. From t2, no task is ready so the idle task needs to run until an event happens that activates Task3 that preempts the idle state because it has higher priority. This task, finishes at t4, the idle tasks takes over and it is activated at t5 again. Shortly after this, the delay timer of Task2 has finished during the execution of Task3. Because Task2 has higher priority, it preempts Task3, which has to wait until Task2 finished in order to continue at t7, and eventually finish at t8. At the end of this example, Task2 starts again but is preempted by Task1 that has the highest priority overall. Task3 might have been activated during this time but is has to wait for both Task1 and Task2 to finish until it can run.

Task Management

Each task is implemented as a C function that contains a endless loop that repeatedly executes a piece of code. In Listing 2.1 a simple example task is implemented with the function `monitoringTask` that reads a sensor value every second and reports if that value surpasses a critical threshold. This start is created in the main-function and after that the FreeRTOS scheduler is started. More tasks could be added by adding a task function and creating it before running the scheduler.

Listing 2.1: Example FreeRTOS task that reads a sensor value every second and prints an emergency statement.

```
#include "FreeRTOS.h"
#include "task.h"

void monitoringTask(void *pvParameters)
{
    /* This loop will never be left */
    for (;;)  {
        // Check if the sensor value exceeds the threshold
        if (readSensorValue() > 200)
            printf("Emergency! Sensor value exceeded threshold.\n");

        // Delay for one second, some other task is run in the meanwhile
        vTaskDelay(pdMS_TO_TICKS(1000));
}}

int main(void) {
    // Create Task
    xTaskCreate(
        monitoringTask,      // Pointer to the task function
```

```
20          "Monitoring Task",    // Name of the task (only for debugging)
21          512,                  // Stack size in words
22          NULL,                 // Pointer for optional parameters for the task
                function
23          1,                    // Priority of the task
24          NULL                  // Pointer to the task handle (not used here)
25      );
26
27
28      // Start FreeRTOS
29      vTaskStartScheduler();
30
31      /* This part will never be reached as FreeRTOS runs continuously */
32
33      return 0;
34  }
```

When a task is created, the kernel allocates memory on the heap for the task stack. Each task gets its own stack, which it uses when it is running. After the scheduler has started, it manages the tasks by tracking which ones are ready to run and orders them based on their priorities. When a task is running, it either sets itself to a blocked state, for example by calling `vTaskDelay`, or it is interrupted by another task. In both cases, the scheduler takes over the control and has to perform a task switch. When that happens, the kernel stores the context of task on top of the task's stack. The context includes all register values of the CPU, the `mstatus` register, that holds general system settings, like enabled interrupts, and the program counter, so the kernel knows where the task was interrupted. It ensures that when the task resumes, it can continue executing exactly where it left off, with all its previous state preserved. Figure 2.16 shows the stacks different tasks at the moment where the running task is being preempted by another task that is ready. It captures the beginning of the task switch when the register values are copied onto the running task's stack. Before the following ready task is run, the copied values from the stack must be first written to the CPU.

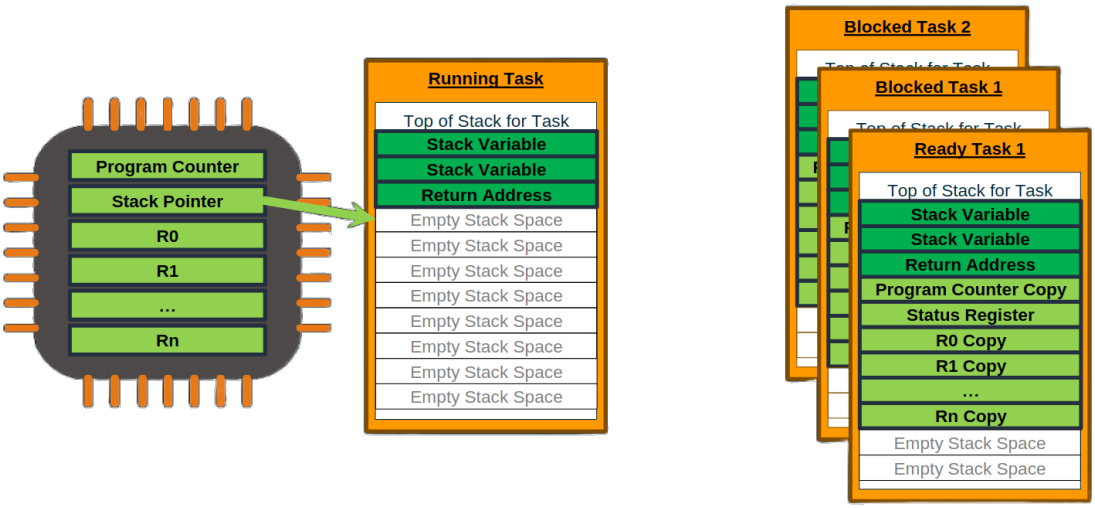


Figure 2.16: Stacks of four different tasks at the beginning of a task switch. The context is stored to the running task's stack, which will be restored before it is reactivated again [43].

2.5. C Toolchain

The human-readable program code written in C cannot immediately be executed by the CPU. As explained before, the CPU executes a series of instructions so it is necessary to translate the C program to a sequence of instructions, also called machine code. This process is far from trivial and consists of multiple steps. In this thesis, the GNU C compiler collection is used, and especially the linker is

leveraged frequently for the implementation of the security enhancements. Therefore, this section is devoted to the C toolchain that provides the reader with the necessary background information.

2.5.1. Preprocessor

The C preprocessor, or `cpp`, is a program that is automatically executed before the compiler to transform the C program. The preprocessor goes through the C source files and scans it for preprocessor directives. The most common directives are `#include`, `#define`, `#ifdef`. These directives mostly perform simple transformations. `#include ``example_file.h"`, for example, literally only copies the content of the file to the location where the directive is put. The `#ifdef foo` keeps the lines of code until `#endif` in the source file if `foo` is defined before, and removes the code from the file if it is not.

`#define` defines socalled macros and these are also used in this thesis to structure the code. If the macro is defined, the preprocessor will replace all occurences of this macro with the value assigned to it. Or if no value is assigned, the preprocessor will remove it (i. e. replace it with an empty string of characters). Listing 2.2 demonstrates the use of this directive.

Listing 2.2: Demonstration of define-directive.

```
1 /* Assign the string '1024' to the macro BUFFER_SIZE */
2 #define BUFFER_SIZE 1024
3
4 /* The preprocessor will transform this... */
5 foo = (char *) malloc (BUFFER_SIZE);
6
7 /* ... into this */
8 foo = (char *) malloc (1024);
```

There are also more complex directives like `#pragma` directives which can influence the behavior of the compiler but these are not relevant for understanding this report. More details can be found in the GNU preprocessor documentatation [44].

2.5.2. Compiler

The GNU Compiler Collection (GCC) [45] is an open-source and popular compiler for C programs and was originally written for the GNU operating system. Nowadays, it also supports many other platforms like ARM, AVR, MIPS, and most importantly for this thesis, RISC-V. Supporting cross-compiling, GCC can compile programs on a personal computer running on x86 for a different platform as RISC-V.

The compiler takes `.c` source files and outputs `.o` object files. Additionally, it also handles `.h` header files that can contain function definitions and variables that are used and shared between source files. Alternatively, `.asm`-files containing assembly code can also be translated into object files. The generated object files contain the machine code but they are not executable yet. In the next step, the linker takes all the object files and puts them together in one executable file.

2.5.3. Assembler

An assembler is one of the important tools in the toolchain of software development for embedded systems. This step comes after the compiler tool and before the linker tool. It converts assembly language, which is a low-level programming language closely associated with machine code, into machine code that can be executed. This translation is needed for embedded systems since they frequently necessitate very efficient and optimized code to function within the limitations of restricted processor capacity, memory, and energy resources. Assemblers help developers create software that interacts directly with the hardware, allowing for precise control over the system's functionality. They are particularly useful in scenarios where maximum performance and minimal resource usage are critical.

2.5.4. Linker

The linker is an important step because it takes the object files that contain the machine code and meta data and composes an executable program. It analyses and resolves the dependencies between the object files. The default file for the executable on many Unix and Unix-like systems executable is the ELF (Executable and Linkable Format) executable. For this thesis, the software will be linked to a

hex-file that can directly be written to the memory of the system. The ELF file is still used because it offers insightful information about the sections of the program executable [46][24].

An executable has at least 4 sections: `.text` contains the code of the program, which are the executable instructions. `.rodata` is the section for read-only data. These are constants that are not altered during runtime. `.data` holds all the variables that are already initialized at compile time. During runtime, the program will read and write to these variables. `.bss` holds all the variables that is not initialized during compile time. During the start of the program, all values of variables in this section are initialized to zero before the execution of the `main()` function.

The programmer can control the structure of the executable with a linker script, an example of which is shown in Listing 2.3. The linker script in this example uses the three commands `MEMORY`, `ENTRY`, and `SECTIONS`.

Listing 2.3: Example linker script.

```
/* Defines the memory regions */
MEMORY
{
    RAM (wxa) : ORIGIN = 0x80000000, LENGTH = 128K
    ROM (rx)  : ORIGIN = 0x00010000, LENGTH = 128K
}

/* Defines the entry point of the program */
ENTRY(_start)

SECTIONS
{
    /* Executable code */
    .text : {
        *(.text)
        *(.text*)
    } > ROM

    /* Read-only data */
    .rodata : {
        *(.rodata)
    } > ROM

    /* Initialized data */
    .data : {
        *(.data)
    } > RAM

    /* Zero-initialized data */
    .bss : {
        *(.bss)
        *(COMMON)
    } > RAM

    /* Custom region for privileged functions */
     .privileged_functions : {
         __privileged_section_start__ = .;
         *(privileged_functions)
         __privileged_section_end__ = .;
    } > ROM
}
```

The `MEMORY` command is optional and it describes the available memory in the target architecture. It is used by the `SECTIONS` command to link it into the correct memory region or to throw an error if the sections do not fit into the size of the memory. If the command is omitted, the linker assumes that there

is sufficient memory available in a contiguous block. The `ENTRY` command specifies the label of the entry of the program. This label should be used once in the source code.

The most fundamental command of the linker script is the `SECTIONS` command. This command structure the layout of the executable file and specifies which section go into which memory space. It describes exactly where the sections are placed and in which order. In the example, the first section is the `.text` which will contain all the executable code. The following section hold the different data sections as explained above. Then the last section is a section called `privileged_functions` which is user-defined and could be a section that shall only be accessible by privileged code. The question is now, how can a programmer tell the linker to put a specific function into this region. This is achieved by using compiler flags. Depending on the compiler this works differently but for the GCC compiler it works by adding an attribute behind the signature of the function as in Listing 2.4.

Listing 2.4: GCC attribute that tells the linker to put the function into the specific section.

```
1 void do_critical_stuff() __attribute__((section("privileged_functions")));
```

This will put the code of this function into the section `.privileged_functions` because it matches the pattern `*(privileged_functions)`. The `*` means that arbitrary characters can be before the section name. It is possible to specify the section even further by adding an attribute to put it into the section `bootloader.privileged_functions` if needed.

The listing above also shows how section symbols are added at the beginning and end of the privileged section. This is useful if the address of the beginning or end is needed in the running program. The programmer alone is not able to determine the addresses of the boundaries of sections, so help from linker is needed. The programmer can use the addresses in the program by marking the symbols as `extern`. The linker will take care of that and assign the correct addresses to these external symbols in the linking step. In Listing 2.5 an example is shown that calculates the size of the privileged data section. It declares the two symbols for the start and end of the section as external because the variables are not defined in the C-code but externally in the linker script. With the `&`-operator, the address of the symbol can be obtained and when the addresses are subtracted from each other, the size is calculated. If an address holds one byte of data, the returned size would be the number of bytes of the section.

Listing 2.5: Determining the size of the privileged section by using the symbols defined in the linker script.

```
1 // Declare the symbols defined in the linker script as external
2 extern uint8_t __privileged_section_start__;
3 extern uint8_t __privileged_section_end__;
4
5 // Function to determine the size of the privileged_functions section
6 size_t get_privileged_section_size(void) {
7     return (size_t)(&__privileged_section_end__ - &__privileged_section_start__);
8 }
```

3

Embedded Systems Security

With the increasing use of embedded systems, their security has become of utmost importance, as the potential impact of security breaches can be significant. This chapter provides a comprehensive background overview of embedded system security. It begins with an explanation of the core security aspects: integrity, confidentiality, and availability in section 3.1. Following this, it discusses physical attacks that target the hardware components of embedded systems in subsection 3.1.1. The chapter then delves into software attacks in subsection 3.1.2, which exploit vulnerabilities in the system's software. Subsection 3.1.3, provides details into network attacks. The chapter then explains four attack scenarios in section 3.2, which serve as the primary focus of this thesis, and concludes with an overview of current countermeasures in section 3.3.

3.1. Overview Security

Security in general deals with preventing an attacker to compromise the confidentiality, integrity, and availability of a sensitive asset. An asset can be any valuable system resource, such as data, software, hardware, or network infrastructure, whose compromise could result in harm to an individual or organization [47]. An attacker, or adversary, is the entity that performs an attack on the system. An attack is an *intentional* assault on the system that compromises an attribute of the CIA-triad: confidentiality, integrity, and availability. Confidentiality ensures that a resource can only be accessed by those that those that are authorized, integrity means that resources can only be modified by those who are authorized, and availability ensures that these resources can actually be accessed when required. Attacks can be either active, by altering system resources or interfere with their operation, or it can be passive, by obtaining information from the system without affecting it directly.

The generality of this definition already suggests that security comes in diverse forms and contexts. Therefore, this chapter will start with a broad introduction on different kind of attacks on embedded systems, attempting to provide the reader a comprehensive overview over this field and the particularities of security for embedded systems. Considering the vast field of embedded security, presenting a system that addresses every potential attack is beyond the scope of this thesis. Therefore, a subset of these attacks that are relevant for the security of FreeRTOS, is examined in more detail afterwards.

3.1.1. Physical Attacks

Embedded devices are particularly susceptible to physical attacks due to their use in often unattended and easily accessible locations. As the number of embedded devices around us is increasing, so is the research for these attacks. Figure 3.1 tries to give an overview of the most relevant physical attacks on embedded devices.

Side-channel attacks do not target the main security mechanism directly but exploit indirect information to compromise the system. These indirect information can be variations in power consumption, or electromagnetic (EM) leaks, among others. For example, by analyzing the power consumption during the execution of cryptographic operations, it is possible to infer the secret keys of many mainstream ciphers

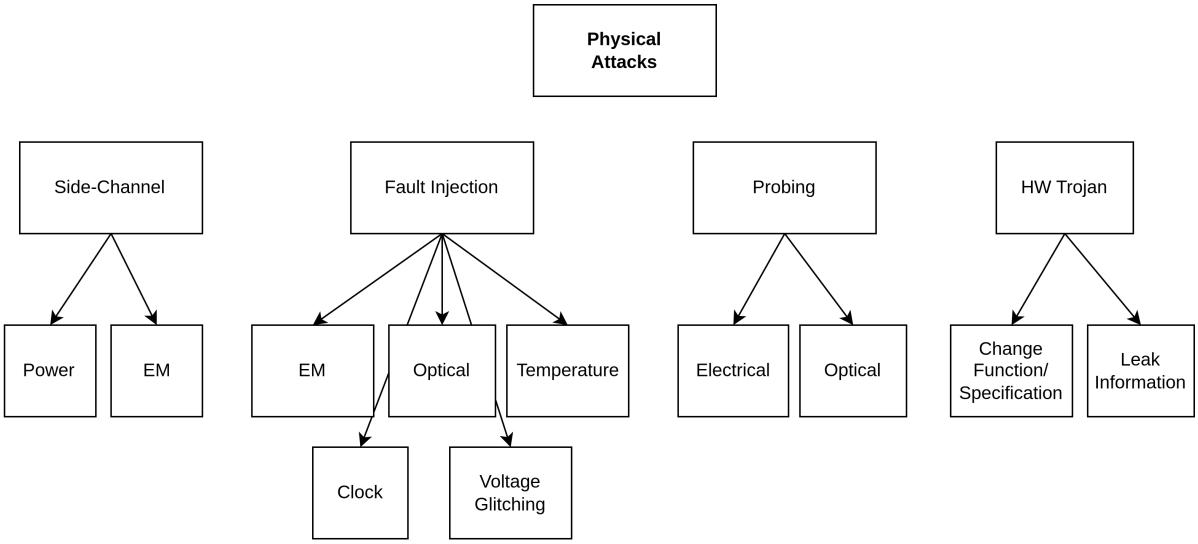


Figure 3.1: Overview of physical attacks on embedded systems.

[48]. For that reason, many security experts plead for a holistic approach to viewing a device's security is required that also take side channel attacks into account [49]. However, many indirect information of the device can offer an attacker valuable information to attack it. Attackers can be highly creative when it comes to inventing novel methods to exploit these subtle leaks of information. These attacks are challenging to anticipate and making a system side-channel proof requires constant vigilance and a comprehensive understanding and analysis of the system. Due to the limited time for this thesis, this work cannot do justice to the challenging task of preventing side channel attacks. Therefore, it is regarded out of scope for this thesis and can be addressed in future work.

Fault injection attacks introduce some disturbance on the system which influence the behaviour of the system [50]. An attacker could bypass security measures, cause the device to malfunction, or extract sensitive information. The injection can be performed with various methods. Using electromagnetic radiation (EM) or light (optical) can, for example, change the content of memory cells, or it can interfere with the operating conditions of the processor that runs the target software. Increasing the environment temperature or the supply voltage beyond the device's normal operating range can lead to malfunctions that can be exploited as well. The same can be happen when increasing the clock frequency: The execution of instruction may be incomplete or skipped entirely, data might get corrupted, etc. The attacks can also be combined. Figure 3.2 shows the setup of a fault injection attack in which a microcontroller is put on a heating plate and the clock signal is manipulated. This way, more faults could be induced and it was possible to repeat individual instructions and to add new random instructions [51]. Fault injection remains a ongoing topic in research because of its diverse injection methods, and each new generation of technology opens new possibilities for attacks [52].

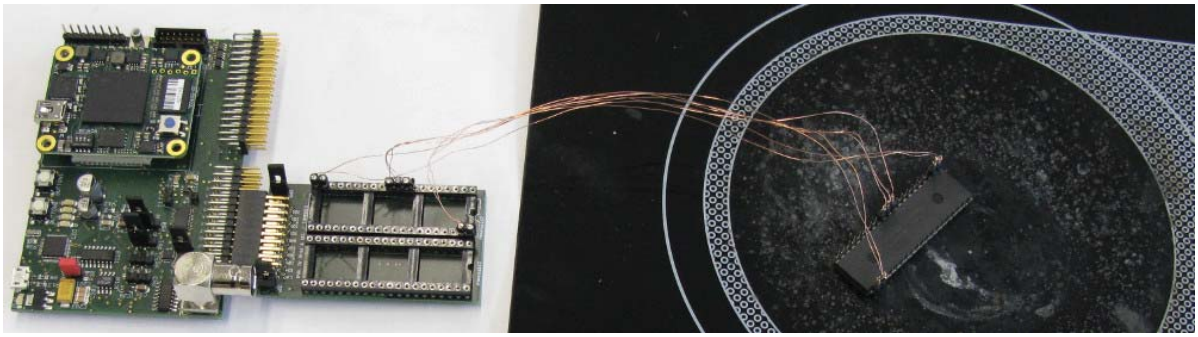


Figure 3.2: Experimental setup for fault injection attack: The fault board is positioned on a heating plate and connected to a microcontroller that induces clock glitching faults [51].

Probing can extract sensitive information from an embedded device. Often, hardware debug interfaces, such as Joint Test Action Group (JTAG) or UART, are available for development and testing, firmware updates, or maintenance. If not secured properly, these interfaces can be an easy entry point for attackers to gain full access to the device. The manufacturer of these devices must disable these interfaces before shipping, or secure them properly at least by adding some authentication procedure, for example with device specific passwords [53]. But even if no debug interfaces are available, with high understanding of the system and identifying target wires, an attacker can access an external memory module, extract on-chip keys, access protected memory, or obtain sensitive device information. However, attacks like can be highly invasive and might destroy the device. Still, for highly critical security devices, influential organizations, such as governmental organizations or terrorist groups, can exploit these vulnerabilities to cause substantial harm [54].

Hardware trojans are malicious malicious hardware components that are inserted into the design, often during the manufacturing process. They are designed to alter the normal operation of the hardware, for example, to enable unauthorized access or to cause failure at critical moments. They can also be infiltrated to leak sensitive data that the device collects during its operation. The trojans can be always active or there are triggered later by various methods. For example, it can be automatically activated after some specific amount of time, or by an external user input that only the attacking party knows [55].

3.1.2. Software Attacks

Due to the growing complexity of their software environments, embedded devices are increasingly vulnerable to software attacks. As the functionalities of embedded systems expand, so too does the potential for sophisticated software-based exploits. Figure 3.3 aims to provide a comprehensive overview of the most significant software attacks on embedded systems.

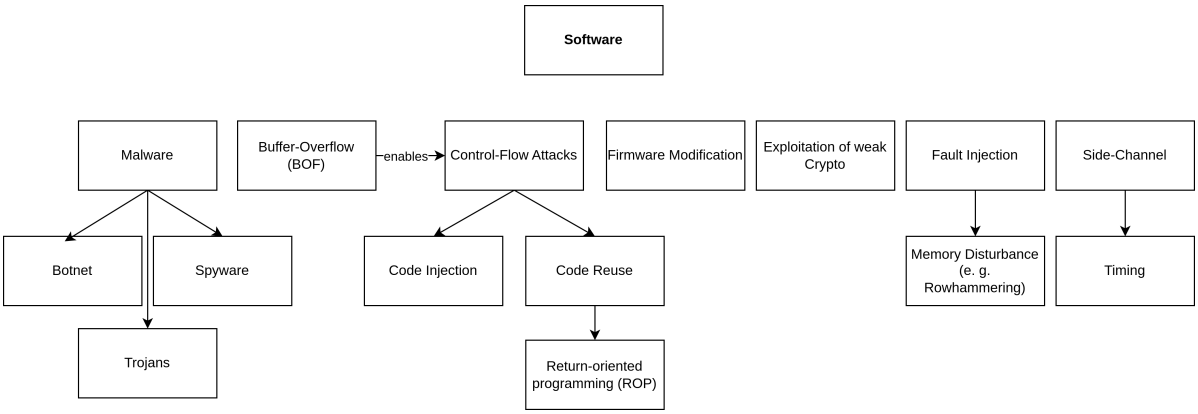


Figure 3.3: Overview of software attacks on embedded systems.

Malware is becoming an increasingly significant threat to embedded systems. In the past, malware was mainly designed for personal computers that run Microsoft Windows, but the rapid increase of the number of IoT devices has now made embedded systems an attractive target for attackers. Along with this development comes that embedded systems often lack security measures, and are often connected to networks. There exist several attacks that scan networks for open ports of poorly secured IoT devices to which the attackers can get access. With enough devices, malware can be inserted that make them act as a botnet to launch Distributed Denial of Service (DDoS) attacks or to mine crypto currency. [56] The Mirai botnet, in 2016, did exactly this and used 65,000 IoT devices to send out so many coordinated requests (DDoS) that service providers, such as Amazon, Netflix, Twitter and others, could temporarily not be accessed [6]. Apart from forming botnets, malware can also be used to exfiltrate sensitive data from the devices or to deploy software trojans to enable or change functionality. Malware does not necessarily need to be injected via the network, it can also be part of third-party software that is flashed onto the device.

Control-Flow attacks are sophisticated family of security exploits where an attacker manipulates the flow of execution of the program. An attacker could manipulate pointer to point to code that performs some unauthorized operation, or to access sensitive data. If the execution flow is redirect to some

other already existing part in the software, it is a *Code Reuse* attack. If the attacker manages to bring additional malicious code into the system that subsequently is being executed, it is a *code injection* attack. These attacks always require exploiting an initial vulnerability that is already present in the system, an Buffer Overflows (BOFs), for example.

Firmware modification can lead to complete control over the device and potentially even remote control. This could be used for unauthorized access, manipulation of the functionality of the device, and data breaches. [57] investigated whether it is possible to read and manipulate the firmware of popular IoT devices that are used in many people's homes. The unsettling finding was that the majority of them do not implement any measures against firmware manipulation. It was possible manipulate the firmware for many smart home devices but also for the network interface of a door lock device, the *Wyze Lock Gateway*.

Weak cryptography or improper use of it can be easily exploitable by attackers. Weak algorithms might be used to save processing time but they can be broken with simple brute force attacks. Similarly, weak or default passwords might be used that allow bypassing security measures although strong cryptography is used. Another weakness can be random number generators that are not seeded correctly. The random number generator often uses a initial seed from which it derives random numbers. If a generator is seeded with the same number at each system boot up, the series of random numbers is predictable and that can be exploited by an attacker. Therefore, ensuring the robustness of cryptographic implementations is crucial for the security of the whole system [58].

Then, there are also software attacks that use fault-injection and side-channel analysis, techniques that we already saw in the physical attacks. These also work in software. Faults can be injected from within the software by using a method called *Rowhammering*. This exploits a hardware vulnerability in memory where repeatedly accessing memory cells can cause cause flips in adjacent memory cells [59]. Timing side channel attack analyze timing variations during the execution of a cryptographic algorithm. With this method, attackers can infer secret keys [60].

3.1.3. Network Attacks

The last category are the network attacks. These attacks can be regarded as higher-level threats because they target the abstraction layers where it is about the data transmission between embedded systems, rather than the physical hardware or the software itself. These attacks are still highly relevant and therefore briefly described in the following. Figure 3.4, again, offers an overview.

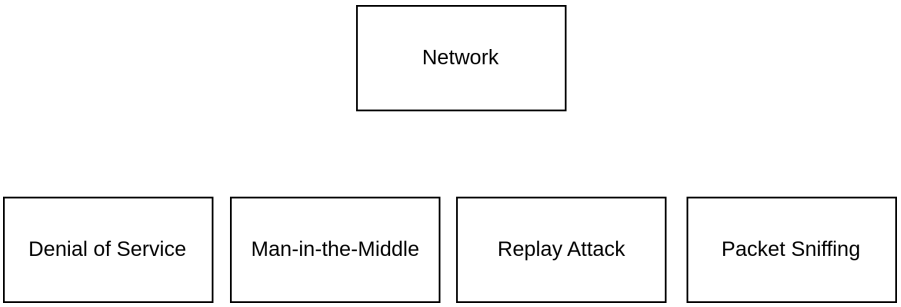


Figure 3.4: Overview of network attacks on embedded systems.

Embedded systems are particularly susceptible to Denial of Service (DoS) attacks because of the low processing power. Many network request in a short time can easily overwhelm the receiving system, hindering it to perform its intended functions. This is exacerbated by the facts that these attacks can be performed remotely, and they are easy to execute because they neither require bespoke hardware, nor in-depth knowledge about the device.

A Man-in-the-Middle attack is a type of attack where an attacker secretly intercepts and possibly alters messages between two parties that think that they are communicating directly. The attacker can eavesdrop on the communication, steal sensitive information, or even inject malicious messages. The communicating parties are not aware that a malicious party is placed between them, hence the name "Man-in-the-Middle" [61].

A replay attack is a type of attack where an attacker records valid packages of the network traffic between two parties and re-transmits packages later to the receiving party again. These packages can contain login credentials or financial transactions. The attacker sends the captured packages later again, or ”replays” them, to impersonate the legitimate user or transaction of the original sender. An example of this attack are early remote keyless entry systems for cars [62]. By capturing the fixed code used to unlock a car and replaying it later, unauthorized access is possible. This attack is often successful in systems that do not use methods to check whether a message is new, like timestamps or random numbers. This makes it easy for attackers to trick the system by replaying old data.

Packet sniffing is a passive attack that observes the data traffic in a network to gain valuable information. This information can be used for subsequent attacks. Attackers scan the networks for passwords, login credentials, emails, or financial details, etc. On unsecured or poorly secured networks, like public networks with no packet encryption, these attacks are particularly effective. Because the attacker does not interfere with the communication between parties, packet sniffing can be difficult to detect.

3.2. Relevant Attacks for this Thesis

As described in the introduction, RISC-V platforms currently lack hardware support to securely isolate FreeRTOS tasks. This thesis presents a design that addresses the security risks associated with this shortfall. Concretely, the design will focus on mitigating control-flow attacks that can be used to execute code that is not part of the task, i. e. attacks from within an application trying to modify outside memory or use kernel functions that it is not authorized to use. The design must assume that the kernel is not modified and can be trusted. Therefore, firmware modification attacks will be studied in more depth and mitigated. Due to the apparent industry wide lack of secure external memory [57], this design will include an efficient design for securing the external memory.

Technology specific attacks, side-channel and fault injection attacks are out of scope for this design because there are to diverse in nature and the design is technology agnostic. Moreover, these physical attacks require bespoke equipment to carry out. The only physical attack addressed in the following is the probing of external memory because it is comparatively simple to do if the wires on the hardware are accessible. Also, the mitigation will mainly look at the firmware level, so application specific malware, network attacks, and third-party cryptography will not be part of this analysis.

3.2.1. Buffer Overflow & Code Injection

Buffer Overflows (BOFs) are often used as a initial vulnerability for other attacks that can happen when the system processes some user input. An attacker would try to input more data than the system is designed to receive. These inputs can be injected via all kinds of interfaces: Serial ports like UART or JTAG, the network interface, human interfaces like a keyboard, etc.

The input has to be stored in an internal buffer in memory, and if not handled correctly, the input does not only write to this buffer but even further into adjacent memory cells. It *overflows* into adjacent memory that is supposed to hold other data. Typically, these attacks occur due to a lack of careful programming practices. Ideally, programmers should always do boundary checks to make sure too large are not written outside of the boundary. However, due to the steady increase in program sizes and complexity, it is not realistic to assume that programmers always manage to do that [63]. If the buffer is on the stack, important other addresses are placed in memory addresses nearby. Figure 3.5 illustrates how a BOF can overwrite the link register, which is the return address of a function that has been entered for performing the input operation. When the function terminates, it jumps back to the address in the link register that holds the return address. If an attacker is able to change this value, it is possible to manipulate the program’s execution flow.

A successful buffer overflow can be used to perform a code injection attack. This attack works in two steps: First, an initial vulnerability is exploited to write executable code into the buffers. This code is a sequence of instructions that have to be well-crafted for exactly this injection. Second, the execution is redirected to the beginning of the injected code. Note, the code has to be written to executable memory. If successful, the attacker gains the ability to execute arbitrary code instructions.

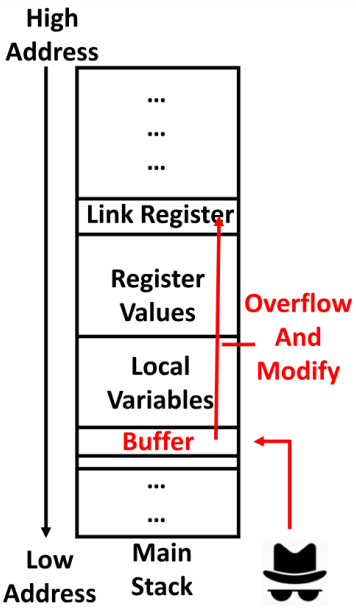


Figure 3.5: Buffer overflow attack in stack frame overwriting local variables, register values, and the return address (link register) [64].

3.2.2. Exploiting Flat Memory Model Vulnerabilities

The flat memory model, characterized by a single address space without inherent separation of user and kernel space, is often used in embedded systems and presents unique security challenges. Compared to the Memory Protection Unit (MPU) and Memory Management Unit (MMU), which were explained in subsection 2.3.3, it enhances the performance and minimizes the area but it lacks security because malicious code has immediate access to the whole platform.

Figure 3.6 presents a simple example of code showing an application-level function inappropriately calling kernel-only functions. In this scenario, the kernel function is directly callable by the application because it is visible during the linking process. The visibility may lead to unauthorized calls that breach the intended security model. Further, there might be instances where the application code is not aware of the presence of a kernel function. In that case, attackers may try to find out the address of the kernel function by reverse engineering. Once the address is known, they can invoke the kernel function by dereferencing its memory address. Since this mechanism of function invocation does not go through traditional call mechanism, it may lead to severe security breaches.

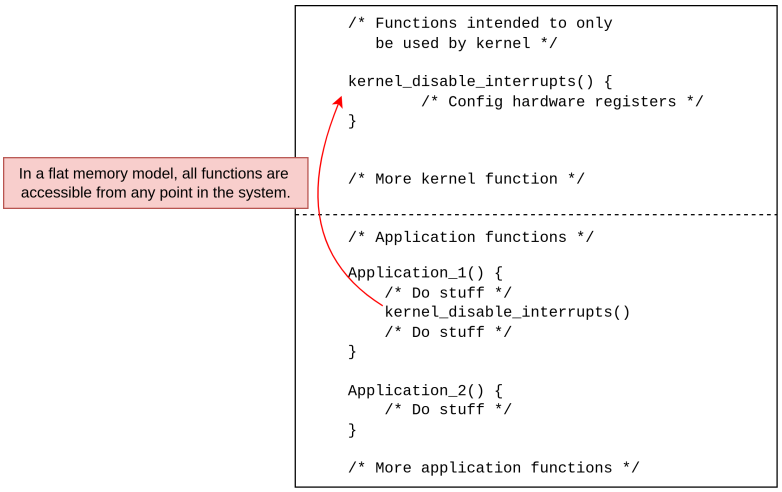


Figure 3.6: Malicious function Application_1 calls function that is only intended to be called by the kernel. In a flat memory model, all functions are located in the memory space. There is no separation of privileges by default.

3.2.3. Probing External Memory for Sensitive Data

When external memory module is used to hold the program and data memory, an attacker can connect to the traces that connect it to the processing unit, and read and write to the external memory unit. The interface typically consists of a data signal and one or more signal for the control. In Figure 3.7 an Amazon Echo Plus, a smart speaker developed by Amazon, was opened to extract the firmware from the external memory [65]. The pins were openly accessible, so by connecting probes to the signals DAT0 and CMD, the memory could successfully be dumped and also written. If an external memory is used, this is not easily preventable because the memory unit cannot distinguish communication to the processor and to an attacker.

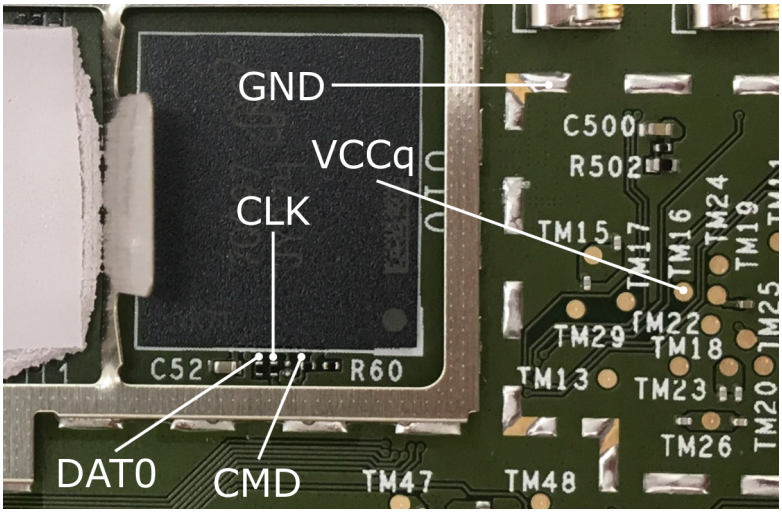


Figure 3.7: The pins of the eMMC chip in an Amazon Echo Plus can be probed to read out or modify data [66].

3.2.4. Firmware and Kernel Modification in External Memory

If writing to the external memory is possible, an attacker can modify the firmware on the device or upload new code completely. The firmware is the one of the most critical part of the software because it configures the hardware and the peripherals during start-up. For that, it needs full control over the platform and runs at highest privilege. After it finished, it hands over control to other software components, in the case of this thesis that is the FreeRTOS kernel. FreeRTOS also needs highest privileges in this design because it accesses the low-level hardware as well. It enables and disables interrupts, it interacts with the timer, manages memory allocation, etc. Thus, modifications on the firmware and the FreeRTOS kernel have the same criticality. An attacker could disable or change features, or add new functionality. It would also be possible to install malware onto the device.

The firmware and FreeRTOS kernel are stored in memory. In the worst-case, these software components are stored in external memory that has accessible pins. For this thesis, only firmware modifications on the memory in rest are considered. That means, that the attacks do not happen at runtime of the system. Attacks on memory at rest are simpler to perform because the stored data is static and there are no read and write requests from the processor. During the active operation of a processor, an attacker would also need to time the write operations correctly to not corrupt the program code. If the timing is miscalculated or imprecise and the write operation coincides with the processor's read operation, it could lead to partial or incorrect updates to the code, resulting in execution errors or unexpected behavior. Therefore, attacks on the memory during runtime are much harder to perform and need bespoke equipment. [67] presents an attack where an malicious memory module holds the modified firmware and an additional controller unit switches between original and malicious memory based on the processor's access requests. Due to the complexity of this attack vector, this thesis focuses solely on static attacks on memory at rest, rather than runtime attacks.

3.3. Existing Countermeasures

These attacks are known and countermeasures already exist. The following subsections describe existing countermeasures to the presented relevant attacks above.

3.3.1. Buffer Overflow & Code Injection

According to [64], code injection attacks can be largely mitigated by making writable memory addresses not executable, also called *Data Execution Prevention*. A simple way for programmers to enforce this is by using an MPU that defines regions that should only be executable *or* read- and writable. Moreover, the injections have to be well-crafted to be executed correctly. Address Space Layout Randomization (ASLR) can make it more difficult for attackers to predict the layout of the memory. Additionally, stack canaries and shadow stacks can be used to prevent buffer overflows. Stack canaries are special values placed on the stack that help detect buffer overflows before they can alter the return address. Shadow stacks are separate stacks used to store copies of return addresses, used to verify the integrity of the return addresses on the main stack.

Buffer overflows mainly occur due to implementation errors. The most languages is embedded systems are still C and C++ [7], both languages are considered memory unsafe and have no inherent mechanism that enforces secure programming techniques. There are memory safe languages, for example Rust, which enforces memory management techniques that prevent the possibility of these vulnerabilities. Code analysis tools can reduce the risk of buffer overflows even for memory unsafe languages by analyzing the source code for implementation flaws. These tools can be very helpful but there is no guarantee that they will find all bugs [63].

3.3.2. Exploiting Flat Memory Model Vulnerabilities

Several ways to prevent flat memory model vulnerabilities exist. A platform that integrates an MMU gives each process its own memory space and can securely separate the accessible memory spaces for each process. For the processes, only their own memory space exist because the MMU lets them work on virtual addresses that are different from the actual physical addresses. This capability must be supported by the OS, which leverages these features. However, due to area, power, and cost overhead this is not feasible, therefore, embedded system typically do not integrate an MMU [68].

Trusted Execution Environments (TEEs) are another way to remedy the inherent flaws of a flat memory model. They are secure areas within a main processor that combine both hardware and software components to create an isolated execution environment. They enhance the security of embedded systems by ensuring the isolation of sensitive data and functionality from the rest of the system. There exist different TEEs for different ISAs and their implementations differ from each other. For ARM Cortex-M platforms, there is TrustZone [69]. TrustZone divides the system into two states: the secure and the non-secure state. The secure state has access to everything on the system, while the non-secure state restricts the access to critical components. This way, user processes can be run in the non-secure mode and the kernel runs in secure mode. Figure 3.8 shows how different components of the systes are assigned to one of the two states. The CPU runs in both states, so it belongs to both regions. The same is true for the SRAM which holds kernel and user data. The other components are assigned to either one of the states on can only be used accordingly. The newer versions of ARM Cortex-M processors also allow to run TrustZone together with an runtime reconfigurable MPU that even enables process specific access rights if the OS supports it without the need of a resource-inefficient MMU [70].

Intel Software Guard Extensions (SGX) [71] is another example of a TEE for x86 processors. SGX allows the creation of secure enclaves, which are isolated regions of memory where sensitive code and data can be executed and stored securely, even if the operating system is compromised. This is achieved by hardware mechanisms that encrypt the enclaves in a way that not even the privileged OS is able to access them. Keystone [20] is an open-source framework for building secure enclaves on RISC-V processors. It provides similar security guarantees by leveraging hardware features to create isolated execution environments, enabling secure computation on RISC-V platforms. Keystone uses RISC-V's PMP feature to isolate the enclaves from each other. These two TEEs are fundamentally different from TrustZone because they follow the enclave isolation approach for different processes while TrustZone focuses more on dividing the system into two states with different privileges. Keystone is the only open-source solution of these three.

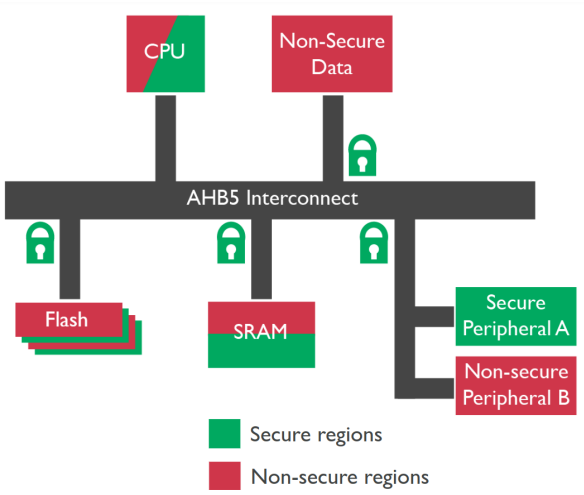


Figure 3.8: Arm TrustZone securing secure regions from non-secure regions [70].

3.3.3. Probing External Memory for Sensitive Data

There are two essential countermeasures against probing external memory attacks. First, storing sensitive data inside the SoC where the processor also resides. This approach eliminates external traces that could be probed. Attempting to access the internal traces would require invasive attacks on the SoC that could also destroy it. Another method, as also suggested by the authors of [65], is to have the processor encrypt the sensitive data before storing them externally. Alternatively, if encryption is not feasible, they advise making the traces at least as inaccessible as possible, for example, by using the inner layers of the Printed Circuit Board (PCB).

3.3.4. Firmware and Kernel Modification in External Memory

There are different possibilities to protect the external memory from being modified by an attacker. The easiest way would be to chose a One-Time Programmable (OTP) memory technology which is secure but would not allow for any updatability. Also, if there is a security vulnerability in the OTP memory, there is no possibility to fix it, other than physically replacing the memory unit. In the following, two more sophisticated countermeasures, the secure boot and external memory authentication, are explained that ensure the integrity of the memory while being flexible and allowing updates.

Secure Boot

The Secure Boot is an often used security measure that verifies the integrity of the memory during boot-up. This prevents an attacker from modifying the program code or data that is stored in the memory of the system. This usually is done with signature schemes that calculate the hash of a memory region and verify it cryptographically [72]. In the simplest case, a secure boot is performed by having some trusted piece of code immediately during the start-up of a system that verifies the program memory that is executed afterwards, see Figure 3.9. If the verification is successful, it is confirmed that the program has not been modified and it will be executed. If is fails, the trusted code does not execute the program because the program in memory is not in the intended state.

For the start-up code to be trusted, there are different ways to accomplish that. One way would be to put the start-up code into OTP. The start-up code is written once during manufacturing and cannot be changed afterwards. This makes the start-up code inherently trustworthy. However, it is also impossible to update this code in case it is needed. For the verification there are also many ways. In this thesis, asymmetric cryptographic operations are used. For that, the owner of the platform has to generate an asymmetric keypair, one public key and a private key that only the owner knows. After the code and data of the original application is compiled to a binary file, the owner calculates a hash of the file and creates the signature by signing the hash with the private key. Having the public key, the hash of the application and the signature issued by the owner, everybody is able to verify the signature of the application. If the binary is changed, the hash changes and the signature verification will fail. This way the integrity of the application can be investigated. It must not be possible to modify or replace the public

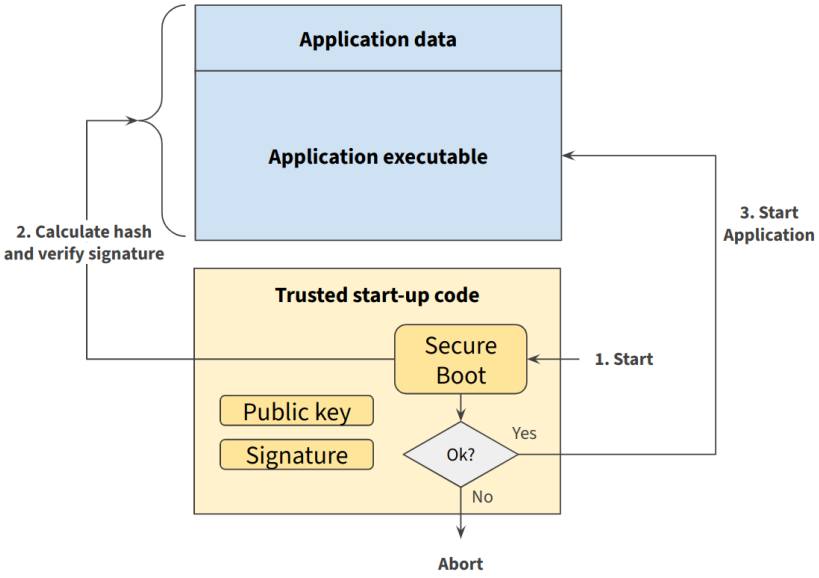


Figure 3.9: Signature-based secure boot procedure.

key by an attacker, because then it would be possible that the attacker generates a keypair and inserts this public key. The signature must not be protected because only the owner of the private key is able to generate valid signatures. Depending on the use case, instead of running the cryptographic hashing and verification as software, higher performance and reduced energy consumption can be achieved by having a dedicated hardware unit for this task [73]. A secure boot can also be implemented much more complex to make it possible to update the code.

External Memory Authentication

Instead of verifying the authenticity only at the beginning during boot-up, the integrity of the data can also be verified during each memory access. In [21], next to the data in memory, there is a cryptographic Message Authentication Code (MAC) calculated with a secret key that is put inside an encryption and authentication hardware unit next to the CPU. During the flashing process, this key is used to generated all the MACs, which are verified during the load operations at runtime. This has the advantages that modifications are detected even during runtime, and it can be used for runtime data memory as well. The disadvantage is that memory access time have a higher delay. This hardware unit also incorporates an encryption unit. Note however, that encryption alone would not protect against modifications. This is a common misconception but encryption does not automatically ensure authenticity. There exist attacks can alter the ciphertext to produce a predictable change in the decrypted plaintext. This can cause the recipient to unknowingly process altered data [74].

4

Secure RISC-V FreeRTOS SoC

This chapter proposes a high-level secure system design that will be the basis for the rest of this thesis. For that, a representative system architecture running FreeRTOS is assumed and analysed for possible attacks. Based on these attacks, requirements are derived that would make the system secure if fulfilled. The requirements are derived in two steps: First, high-level cybersecurity goals are defined that specify a need while being solution agnostic. In the second step, specific feature requirements are developed that form the foundation for the design in the following chapter.

On a high level, embedded systems integrate a CPU, memory, and peripherals (see section 2.1). A System on Chip (SoC) is an integrated circuit that consolidates all of these components onto a single chip. With the rise of mobile and embedded devices over the past few years, the use of SoCs increased due to lower power consumption, better cost-efficiency, reduced design-complexity, and higher reliability by avoiding soldering issues or connector problems. The security of the system also increases because the components all reside inside the SoC and are less accessible than on a “system on board”, where the components are connected via traces and are vulnerable to probing attacks [75]. In small embedded systems, the memory footprint might be so small that it can fit on the on-chip memory inside the SoC. In this case, an attacker would have to spend significant effort to break open the package to get access to the internal memory. For larger, more complex embedded systems, there often are external memory modules used in which the program or parts of the program are stored. In this case, the attacker has full access to the memory and can interact with it in the same way as the processor. That means the attacker can read and write any address of the external memory. This chapter investigates the security of a SoC representative for use in embedded devices that incorporates a flat memory model and uses external memory, by focusing on the relevant attacks described in the previous chapter. In the following, the state of the art is presented, which explains what others have done to secure FreeRTOS on devices with a flat memory model. After that the system architecture for this thesis is developed.

4.1. State of the Art

Recent research and advances in the field of secure embedded systems have brought to light a variety of techniques that may be utilized to handle security issues. This is especially true with real-time operating systems (RTOS) that are extensively utilized, such as FreeRTOS. Alternative solutions have been investigated by researchers such as Lindemer et al. [76] , Thomas et al. (2018) [77], and Garlati et al. (2021) [78] as a result of the inherent security vulnerabilities that are present in FreeRTOS. In order to improve the level of security, these methods make use of Trusted Execution Environments (TEEs), as opposed to directly executing the RTOS kernel with the maximum possible privileges. RTOS tasks are executed at an even lower privilege level in these proposed systems, while the TEE kernel functions at the highest privilege level. The RTOS kernel is demoted to the second highest level, while the TEE kernel runs at the highest level. This design not only makes the system more complicated, but it also necessitates the use of a platform that is able to handle three unique privilege levels. The highest privilege level is reserved for the TEE kernel, the second privilege level is given to the RTOS

kernel, and the third privilege level is given to RTOS tasks. Support for such a structure is included in the RISC-V Instruction Set Architecture (ISA), also known as the RISC-V Privileged Specification [35]. This architecture is often utilized in “Unix-like operating systems”. TrustLite [79] addresses this issue and presents a TEE that only uses two privilege levels, being more suitable for embedded applications. It secures the flat memory model architecture by using an *Execution Aware* MPU (EA-MPU). This is an MPU that is not actively controlled by the CPU but it controls itself based on the program counter (PC) of the CPU. The CPU configures the EA-MPU once at the beginning of the program. This avoids that the kernel has to be adapted to dynamically reconfigure the MPU during each task switch. However, this approach is very inflexible because the EA-MPU can only be configured once at the start-up, and the configuration is stored in hardware which imposes strict limitations on the software, forcing it to operate within these predefined constraints. Also, this EA-MPU is a non-standard hardware unit, which may face challenges in gaining widespread adoption in the market. Using FreeRTOS with full privileges increases the risk of exploiting kernel security flaws to gain system-wide access [80]. On the other hand, recent enhancements made by the authors of FreeRTOS have considerably improved the kernel’s security, and the kernel’s modest size automatically decreases the attack surface. The industry has not yet adopted more secure versions of real-time operating systems (RTOS), despite the gains that have been made. It is common for people to be hesitant because they are concerned about their profitability and the dangers that are connected with shifting to new systems that have not been tested, even when there are obvious benefits in terms of security [81]. It is necessary to perform a comprehensive rework of both the software and the hardware in order to integrate FreeRTOS into a TEE. This might be an undesirable and unrealistic option for many developers and businesses. The purpose of this thesis is to achieve a balance by boosting security in energy-restricted embedded systems without requiring the adoption of sophisticated TEE designs. This will result in a solution that is more viable and favorable to the industry.

4.2. Requirements for Secure SoC

This thesis will propose a system that leverages the PMP mechanism of FreeRTOS to enhance the security. Before, the design and implementation is presented in the next chapter, the next part will explain the threat scenario that this system protects against. For this, first attack paths are described that are relevant for this thesis. From these attack paths, requirements for a secure system are derived. The requirements derivation is performed in two steps. In a first step, high level Cybersecurity Goals (CGs) are defined that does not describe a solution yet but define a more abstract requirement. Eventually, these high level requirements are further developed into concrete security measure that can realize a CG. This analysis aims to explain which attacks are prevented with the integration of PMP support for FreeRTOS, and which other measures have to be installed to secure the system as a whole. The PMP support is only one piece of the systems’s security, albeit a significant one. Still, when implemented in a real-world application, it must be complemented by other measures.

4.2.1. Attack scenarios for Embedded Systems

For this thesis, we assume that all program code and data, as well as the Random Access Memorys (RAMs) resides in the external memory. With that, also the stronger configurations with parts of the program being stored on-chip are covered. The high level system architecture and the Damage Scenarios (DSs) DS1 to DS4 that an attacker can perform are presented in Figure 4.1 and explained in the following.

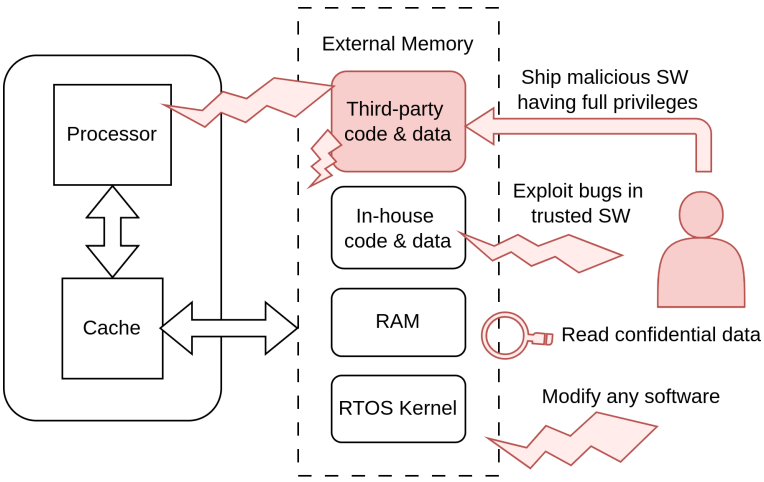


Figure 4.1: System with external memory and the possible attacks

DS1: Ship malicious SW having full privileges

An attacker could provide third-party software to a platform that can compromise the whole system. Because typically a flat memory model is used, the shipped software can read and write other data, as well as tinker with the internals of the system, for example deactivate interrupts. For this attack, the attacker does not even need physical access to the device. Instead the attacker can provide seemingly benign code that incorporates malware. For example, the malware could eavesdrop on the confidential data of the system's owner via a network interface, or it could take take over control of motors that are controlled by another program on the same platform. Once malicious software made it into the memory countless attacks are conceivable, compromising the system's security to the detriment of the user's privacy and safety.

DS2: Exploit bugs in trusted SW

An attacker could also exploit software vulnerabilities of the trusted platform from the platform itself. Especially if the software has was poorly designed, it might be prone to BOF attacks. Or if it is a networking application, it might have to deal with packets of variable length. Without measures for secure handling of input data, code injection attacks might be possible. In both cases, the trusted code could read or write to memory regions outside of its assigned region, which results in the same harm to the owner as in the previous attack.

DS3: Read confidential data

Given an attacker has access to the traces that connect the CPU with the external memory module, the attacker can arbitrarily read out data, including confidential information that can violate the owner's privacy or extract IP.

DS4: Modify any software

Again, given the access to the traces, an attacker could change the systems behavior. This could be done by injecting executable code, change parameters, control the program flow, or manipulate the CPU's status and control registers. In the simplest case, this could result in the system to shut-down, but it could also lead to life-threatening situations if the system is, for example, responsible for steering a car.

4.2.2. Cybersecurity Goals

The derivation of the CGs is shown in Figure 4.2. A CG is a high level requirement that addresses the damage scenarios. This is useful because it describes the concept of the security on a high level. If these requirements are fulfilled, the system is considered secure, regarding the selected damage scenarios. The CGs can be accomplished in many different ways, but it adds some clarity to first state them and derive concrete requirements in a next step.

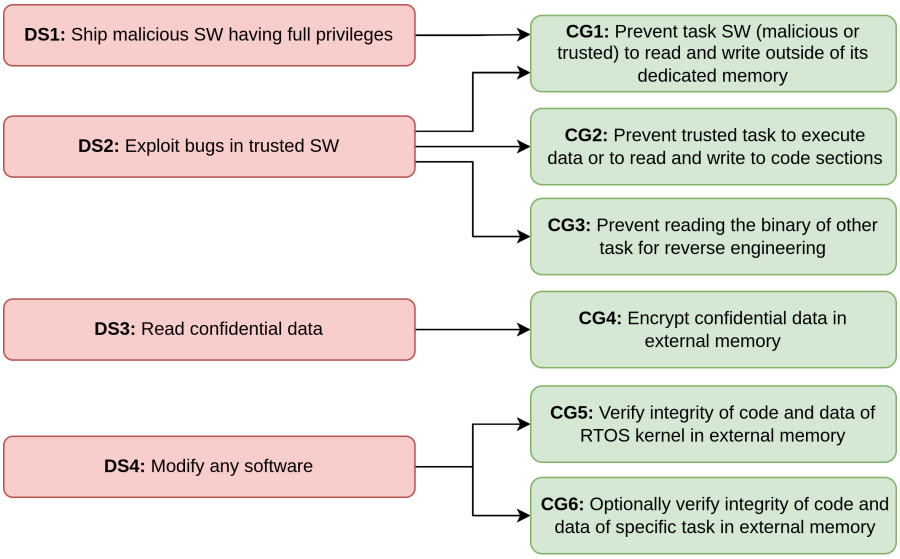


Figure 4.2: Deriving Cybersecurity Goal (CG) from Damage Scenario (DS).

To mitigate DS1, it is necessary to restrict a task to only the memory and peripherals that it is authorized to use (CG1). This comprises the executable binary of the task itself, its data section, the memory for its runtime stack, and peripherals as well as dedicated shared memory that the task uses. The access to anything else should be denied. To mitigate DS2, the same restrictions as in CG1 shall be applied also to trusted tasks. However, this is not sufficient because code injection attacks on trusted tasks would still be possible. An attacker could provide some legal input to the task, for example via a network interface or a read buffer. Conceivably, exploiting weaknesses in the trusted SW, BOF attacks could inject executable code into the task's data. This way, an attacker can compromise the task and the peripherals that the task has access to. To prevent these kinds of attacks, the code sections shall not be read- or writable and the data section shall not be executable (CG2). Furthermore, to prevent these vulnerabilities to be found in the first place, the executable binary of the task must be protected from being read. This prohibits an attacker from reverse engineering the task and finding vulnerabilities. DS3 can be prevented by encrypting the external memory in a cryptographically strong manner (CG4). DS4 is prevented by verifying the integrity of the external memory. This includes the start-up/initialization code, the RTOS kernel, and the tasks' code and data. Given that the previous CGs are ensured, it can be argued, that tasks that do not use critical peripherals do not necessarily need to be secured. For that reason, continuously verifying the confidentiality for a specific task is optional, as on small embedded systems, this often is an time and resource consuming process.

4.2.3. Derived Requirements

The CGs can be achieved in various ways. This section presents which measures this thesis implements to make the system secure. Figure 4.3 summarizes the requirements. In the following, these requirements are explained in more detail.

The first CG to prevent tasks to read and write outside their memory regions is realized by leveraging RISC-V's Physical Memory Protection (R1). Each task shall be assigned memory regions, at minimum the code binary of the task and the task's stack memory. Additionally, other regions can be added, for example, for shared memory between tasks or memory-mapped peripherals. The tasks shall not be able to changed their or other tasks' PMP rules. The kernel shall be the only component that can modify these rules. The second CG is achieved by setting the appropriate read/write/execute permissions in the PMP-configuration for each region (R2). Code sections shall strictly only be executable and data sections only read- and writable. Again, the only the kernel shall be able to change the settings. To prevent reading the binary for reverse-engineering and reading out confidential information, an external memory encryption unit in hardware shall encrypt data that leaves the CPU for specific tasks (R3). The memory encryption unit must be implemented in hardware and the key shall not be accessible by the software running on the CPU. The RTOS kernel is the most privileged software component

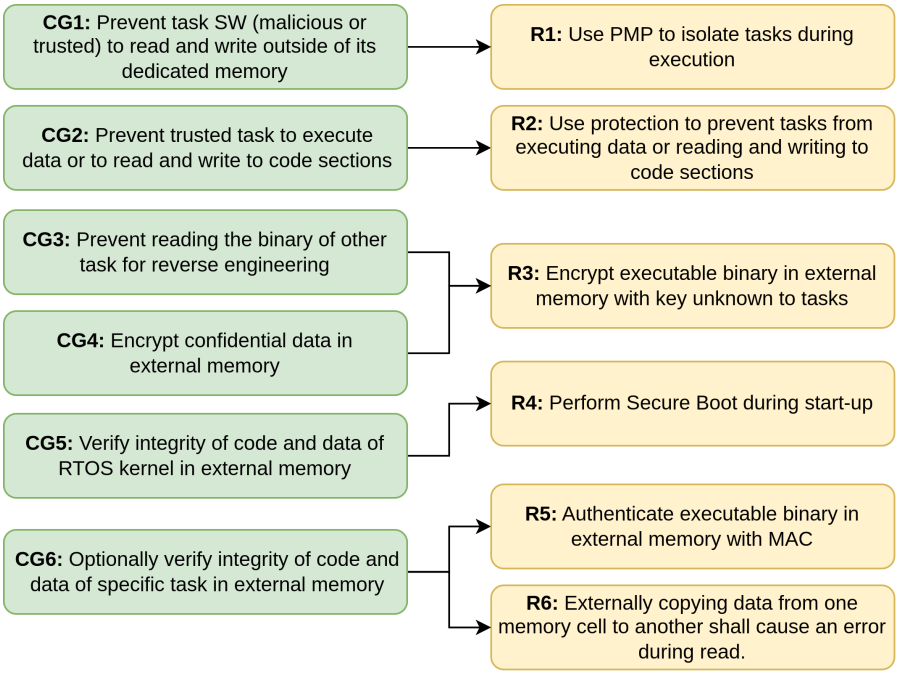


Figure 4.3: Deriving requirements from Cybersecurity Goal.

of the system. It must be ensured that the kernel has not been altered, otherwise an attacker could change the binary in a way that memory protection measures are not enforced. A secure boot shall be implemented that verifies the integrity of the kernel and all called code that is executed before (R4). A secure boot on embedded systems is usually an intensive operation when done in software. The secure boot calculates a cryptographic hash over a region of memory and the bigger the region, the longer the execution time. The start-up time can be drastically decreased if only the kernel is verified, instead of the whole application, including the user tasks. This is acceptable because the tasks are running at lower privileges and restricted by PMP, and they can also be checked for integrity by the next requirement. Optionally, it should be possible to verify the integrity of specific user tasks in the external memory. The external encryption unit should also be able to calculate and verify the MACs of memory blocks. The key that is used by the encryption unit for encryption and the authentication with MACs shall only be known to the vendor and the encryption unit. This way, it can be ensured that the data in the external memory is either issued by the vendor or written by the CPU on the system. Given that the previous requirements are fulfilled by the system, the tasks are more restricted to their memory region and peripherals. If the peripherals it accesses do not allow for serious harm, e. g. a temperature sensor that only provides a value, it can be reasoned that the task does not need to be secured by MACs or even encrypted. This is useful again to reduce the processing time on cryptographic operations. The last requirement is more subtle. Assuming the memory in the external memory is encrypted and authenticated, it must not be possible to copy data from one cell to the other. Normally, the MAC would be stored next to the encrypted data. Conceivably, an attacker could copy the content including the MAC and put it to a different address. A possible attack path would be that a malicious application writes some data, that it wants to inject into a different memory region. to the external memory. By reading the external memory, the encrypted and authenticated data can be obtained that the attacker could write to a different task's section. To prevent this, the encryption process shall securely include the address of the memory block (R5) to make sure the data only works at a specific address.

4.3. High-level Proposed Secure System

Figure 4.4 showcases the enhanced system overview with the security improvements from the requirements. It shows FreeRTOS running two different tasks. Each task has its own code section, i. e. the executable binary of the task, and its own data section, i. e. task stack and variables that are initialized at compile time. Task 1, in this example, uses the network interface, so the PMP configuration

allows task 1 to access its own code and data memory and the memory-mapped network interface. Task 2 does not use peripherals, like the network interface, and can thus only access its code and data section. Then, there is also the unprivileged section that is accessible for all tasks. These are typically standard library functions and FreeRTOS system calls. In this example, these are the standard C-library functions but it can include any code that does not need higher privileges and is considered safe. The system calls are needed for the tasks to interact with the kernel. If the task needs to perform an operation that needs to run on higher privilege, the task request the kernel to perform the operation on behalf of the task via these system calls.

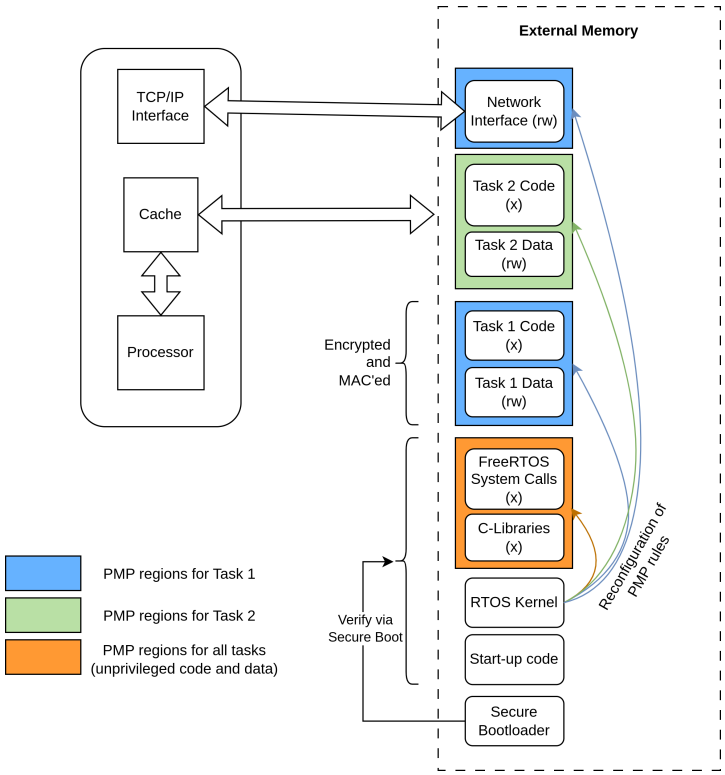


Figure 4.4: Proposed system with enhanced security.

Additionally, tasks can be optionally encrypted and secured with a Message Authentication Code (MAC). A MAC is a cryptographic tag to a piece of data that proves the integrity of the data. In this example, task 1 is secured this way because it has access to the network interface, which an attacker could compromise when modifying the code and/or data of task 1. On the contrary, task 2 is considered less critical and is not secured. The decision whether or not to secure a task this way should be made after a careful analysis of the harm that a task poses to the system after being compromised. The advantage of not securing a task is the time and energy savings for the encryption and authentication operation. Everything mentioned before only works effectively if the underlying kernel and code are protected against tampering. A secure boot verifies the integrity of the binaries of all software components up until and including the FreeRTOS kernel. It is important that the secure bootloader that executes the secure boot is trusted and has not maliciously been modified. This can be achieved by setting the memory region of the bootloader to OTP.

The design and implementation of this proposed system is the topic of the next chapter. All presented features are implemented, except for the optional task authentication because the time needed to complete it would not be worthwhile, as it is very similar to the encryption module and would not add much new knowledge.

5

Design & Implementation

This chapter presents the design choices and implementation of the secure system presented in the previous chapter. It begins with the evaluation of different available RISC-V cores and selects the one the fits the needs of this design best in section 5.1. This is followed by the memory layout and hierarchy of the design in section 5.2. After that, the secure boot to ensure the integrity of the FreeRTOS kernel is presented in section 5.3. Eventually, the design and implementation of using RISC-V's PMP feature FreeRTOS to restrict tasks, is presented in section 5.4. Lastly, the combination of the memory encryption unit with the PMP feature to selectively encrypt FreeRTOS tasks is presented in section 5.5

5.1. RISC-V Core Implementation

The CPU is the central component of an embedded system, acting as a brain that controls and manages all operations. The goal is to implement a platform based on RISC-V, which requires selecting an appropriate CPU core for the design. However, there are countless ways to implement a RISC-V CPU because RISC-V, the ISA, only specifies the behavior of the core but it does not prescribe how it should be implemented. In the following, different open-source cores are presented and the most suitable RISC-V core is chosen. The selection is based on how well a CPU-core fits the requirements, which are:

- Supports PMP, preferably up to 64 PMP-regions
- Representative for typical embedded MCUs (see section 2.1)
- Maturity

5.1.1. Available RISC-V Cores

There are many RISC-V cores that are regularly used in literature. The UC Berkeley, where RISC-V was invented, developed two synthesizable RISC-V cores. First, the Rocket-Chip [82], being rather a core generator than a core itself, and the Berkeley Out-of-Order Machine (BOOM) [83]. The Rocket-chip can generate highly-customizable 64-bit 5-stage in-order RISC-V cores. Making it customizable is attractive for academia because it allows for extensive design explorations. It supports the full RISC-V ISA which means that it is possible to configure a core with up to 64 PMP regions. BOOM is a synthesizable open-source RISC-V out-of-order core aiming at high performance. Its design is more complex implementing features like out-of-order execution and branch prediction.

Another popular core is the CV32E40P [84] that was formerly known as RI5CY. RI5CY had been maintained by the PULP platform and was contributed to the OpenHW Group in 2020. This 32-bit core aims at being small and efficient, being in-order RISC with a 4-stage pipeline. It also implements PULP custom extensions for achieving higher code density. Focusing on efficient performance, the core does not support PMP at all. OpenHW Group also developed the CV32E40S [85] which belongs to the same platform family but it specializes on security rather then performance, indicated by the S for security at the end of the name, instead of the P for performance. The CV32E40S is also a small 32-bit core

with 4 stages but it can implement up to 64 PMP regions, depending on the parametrization. It supports supports two privilege modes, M- and U-mode instead of only M for the CV32E40P. Additionally, it implements a custom extension called *Xsecure* that enhances the core's security with features like dummy instruction insertion, checksums for register values, and data independent timing, which aim at making side-channel attacks more difficult. The VexRiscv core [86] is a highly customizable 32-bit RISC-V core being often used in research for its versatility, similar as the Rocket core. It is a 5-stage core, it implements 3 privileges and it has full PMP support. Apart from that, the platform offers many plugin that can enable the core for branch prediction, multiplication/division, floating-point or others. Alternatively, there are PicoRV32 [87] and the CV32E20 [88] (former Zero-RI5CY) that are small RISC-V cores that try to be as minimalistic as possible. They target applications in low power and low resource environments, or are used for educational purposes due to its simple design. The PicoRV32 uses a 1-stage pipeline and is smaller in area and performance then the 2-stage CV32E20. However, the minimalistic design choice forbids the use of PMP and neither offers multiple privileges.

5.1.2. Evaluation of Cores

Based on the requirements, this section explains which core is chosen for this thesis and why. Finally, Table 5.1 shows a comparison of the presented cores with a qualitative rating.

The Rocket-Core incorporates PMP, is mature as it has been around for many years and experienced multiple iterations. However, being a 64-bit platform makes it score weak in representativeness. It is highly customizable so it could be changed in a way that it is more suitable for resource constraint embedded systems, nevertheless, 64-bit is rather unusual in this domain. BOOM scores similar to the Rocket-Core with the difference that it is both more complex and less adaptable which makes it even less representative. The CV32E40P is perfectly representative for the use case, however, it is not equipped with PMP which is a criterion for exclusion for this thesis. The CV32E40S scores better in this regard because it is equipped with 64 possible PMP regions which makes it an excellent fit for this thesis. One downside is that this core does not have the same maturity as the others. The CV32E40S is a recent project that emerged from the CV32E40P and had its design freeze at the beginning of 2024. Nevertheless, the documentation is well maintained and many components are took over from the other more mature cores of the OpenHW Group CORE-V family. The VexRiscv also is an excellent fit because it supports PMP, is a resource-efficient 32-bit core, and has a high degree of maturity. The PicoRV32 and the CV32E20 both do not support PMP and, therefore, score very poorly. The PicoRV32 is also too simple and minimalistic that it is not representative for the platforms that are discussed in this thesis. The CV32E20 could be used for somewhat demanding applications and could be interesting for this thesis, but the lack of PMP support makes them not well suited for the SoC.

Table 5.1: Comparison of existing RISC-V cores suitable for FPGA synthesis.

Softcore	PMP	Represenatitve	Maturity	Remarks
Rocket-Core	+	-	+	Highly configurable
BOOM	+	-	+	-
CV32E40P	-	+	+	-
CV32E40S	+	+	O	Security focus
VexRiscv	+	+	+	Highly configurable
PicoRV32	-	-	+	-
CV32E20	-	O	+	-

Based on the analysis, **the CV32E40S is chosen for the design**. It supports the maximum number of PMP regions, it fits well into the context of an embedded system, it is reasonbly mature, and it implements additional hardware security features. From the table, the VexRiscv ranks similarily and is even more mature, however, the security focus of the CV32E40S makes it interesting for future advancements. Also, the CV32E40S was released comparatively recently, well-established RI5CY core, which is still regarded as a mature and reliable design. After having selected the RISC-V core, the next section will explain the components around it: the memory and peripherals.

5.2. Memory and Peripherals

The chosen core is integrated into a SoC that uses external memory for all code and data, and integrates various peripherals. Figure 5.1 presents the whole design of the system. The core is connected to the external memory via two datapaths: the instruction fetch path that accesses the code from the external memory for the fetch stage, and the load-store-unit path for data read and write operations that are requested during the execute stage. Both paths feature a cache and an encryption and authentication module. There are two separate modules for each path instead of having both shared because the fetch and execute stage will be active at the same time due to instruction pipelining. If they used the same hardware units, they would potentially block each other. The memory arbiter manages simultaneous access requests and data accesses are prioritized.

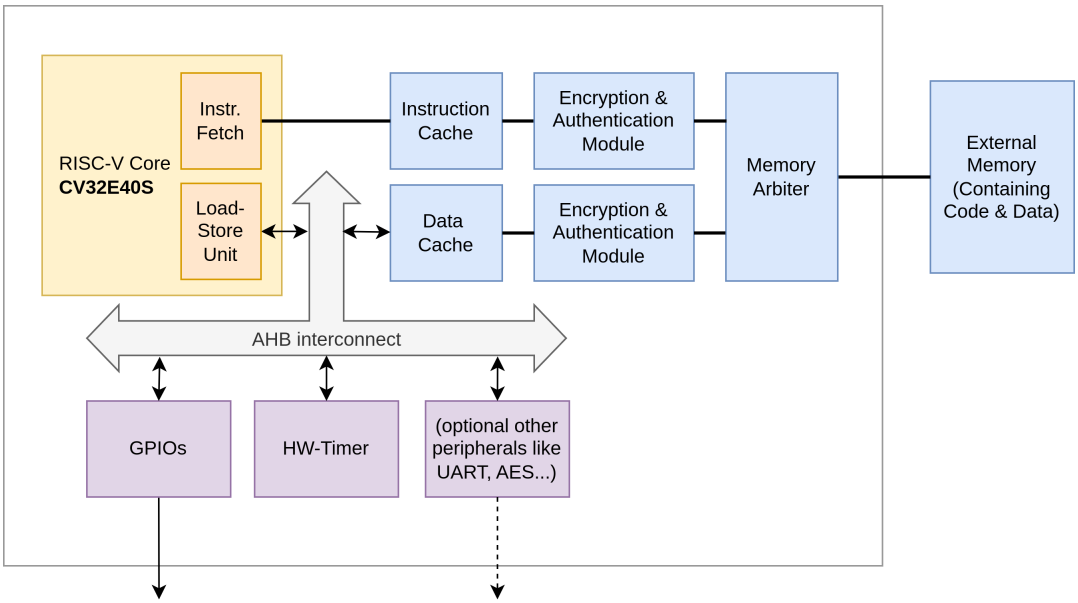


Figure 5.1: Final System-on-Chip (SoC) connected to external memory.

The the load-store-unit is not connected to the data cache directly but accesses it via an AHB interconnect. The reason for that is that the peripherals are memory-mapped to the address space and the AHB interconnect provides the needed hardware logic for that, as explained in subsection 2.3.1. AHB is designed for high-speed communication in SoC designs [27]. It connects the core to the data memory, the GPIO peripheral, and the HW-Timer. This selection of peripherals is minimal and can be extended by more peripherals. The GPIOs are used to have some simple interface to the system. Four of the the GPIOs are connected to LEDs on the board that can be useful to show the status of the system. The timer is important for the SysTick-interrupt for FreeRTOS which is mandatory for the scheduler.

5.2.1. Memory Layout

The memory layout is organized in sections as shown in Figure 5.2. It starts at the address 0x1C00_0000 with the sections that are put into the external memory. In the physical memory, it starts with the address 0x0, so the offset of the starting address has to be subtracted in order to obtain the physical address. These sections do not need to follow the strict order and can also be arranged differently. The top sections are memory-mapped peripherals and will not end up in the external memory. They are connected to the peripherals via the AHB interconnect as described in section 5.2. The memory layout starts with the section for the key material. This section contains keys used for the cryptographic operations. This section is special in a way that it needs to be stored in some secure memory block, preferably inside the SoC and not in the external memory. It is a small section that holds a few keys that must be protected against unauthorized modification and reading. Then, two sections for the privilege data and code (functions) follow. These sections holds memory that should only be accessible to the kernel. Then, there is the section that contains the system call functions with which the code can request services from the kernel. It is marked as unprivileged to grant user tasks access to these functions. For security

reasons, system calls are only allowed to be called from these last two code sections. When a system call is called, the corresponding function in this section will execute an `ecall` (environment call) which will start the trap handler. The trap handler will verify whether the call originates from these sections by investigating the return address, which is `mepc` (Machine Exception Program Counter). This prevents that tasks can perform an `ecall` and request services directly. The kernel should always be able to issue `ecalls`, therefore these two sections are marked by linker directives as the address range from which system calls are valid. The reason why these sections are put at the beginning of the layout is solely because they are fixed in size, independent of the application code, which is convenient during the development because these functions do not grow and move around the following sections when application code changes.

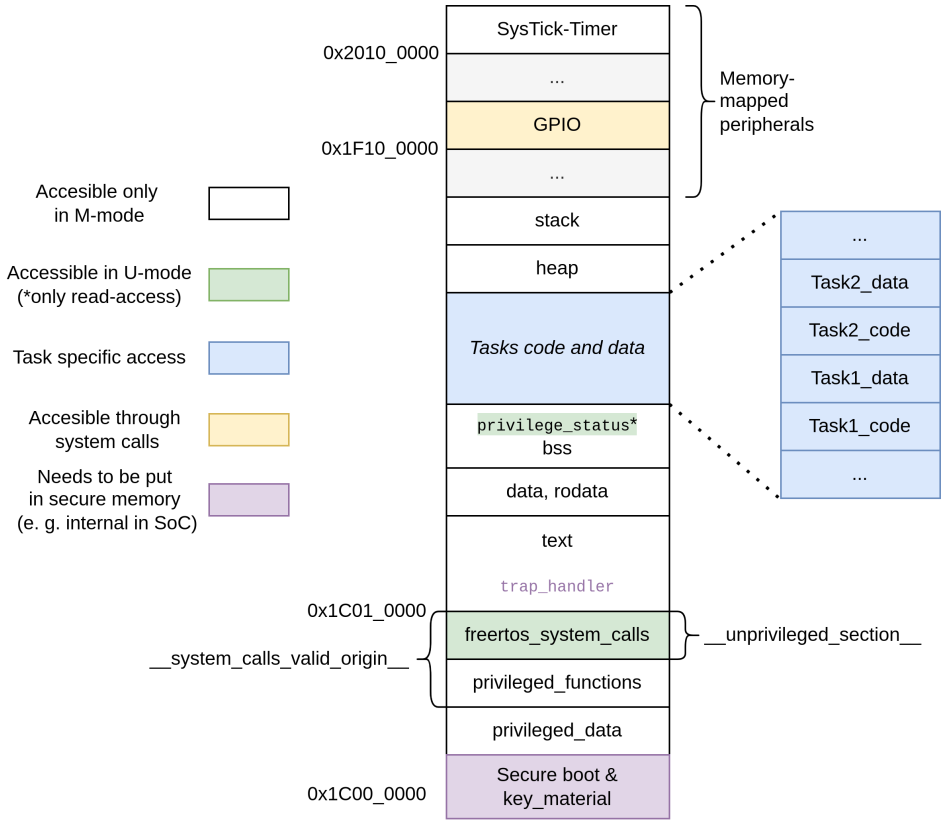


Figure 5.2: Layout of memory space for this implementation including access rights. The regions at the top are memory-mapped peripherals and the rest is located in the external memory unit.

The next section is the `.text`-section, that is the standard destination for program code. At the beginning of this section resides the trap handler from the previous section with the entry at address 0x1C01_0000. At the beginning of the execution of the platform, this address must be written to RISC-V's `mtvec` register, so the CPU automatically jumps to this address in case of a trap. Then, the standard sections for the data follow: `.data` and `.rodata`, here summarized as one section, followed by the `.bss` section. The last 32-bit of this section hold the read-only `privilege_status` variable. After that, the code and data for each task follow. They are arranged according to the task encapsulation scheme described next in subsection 5.2.2. In normal FreeRTOS tasks, the stack is allocated on the heap during task creation. In this implementation, the stack is statically allocated to the task-specific data section before task creation. These regions are protected by the dynamic task-specific PMP regions. Lastly, the heap and stack follow, that are used to allocate memory by the FreeRTOS kernel. This could also be used for additional memory regions for sharing data between tasks. The addresses for these shared memory ranges would need to be added to the tasks' PMP configuration at task creation.

5.2.2. FreeRTOS Task Memory Encapsulation

For this approach to work, the function and subfunctions of a task need to be placed in adjacent memory regions. Otherwise, it is not possible to capture the address range with only two PMP-regions. For this, the linker script is leveraged to control where pieces of code will end up in memory. After the tasks are compiled, each task's executable, i. e. the code, is put in its own sections with labels marking their beginning- and end-addresses. This way, the program has access to the addresses of the beginning and end of the task code and data and can easily use it to configure the task specific PMP-regions. The code and data are put in different section for two reasons: First, the linker does not allow code and data in the same section, and second, the PMP-region would be configured with *rwx* permissions, which is not secure because that would allow data to be executed. Therefore, two regions for one task are needed for only the task itself.

The C toolchain of preprocessor, compiler and linker are agnostic to the concept of FreeRTOS tasks. On that level, a task is simply a set of functions, and the linker is free to chose where to place these functions. As explained in section 2.5, by default the linker would the code of all functions of a task in the `.text` section and the data in the data sections: `.data`, `.rodata` or `.bss`. This is not beneficial for the task encapsulation because the functions of different tasks are mixed together in these sections. Figure 5.3 illustrates this. The left side shows the default case where the linker puts the functions into memory in an unordered manner. The linker has no concept of tasks, therefore, it simply inserts functions and data into the according sections. With the linker script it is possible to group functions and data structures that belong to one task, such that they are put in a contiguous memory section.

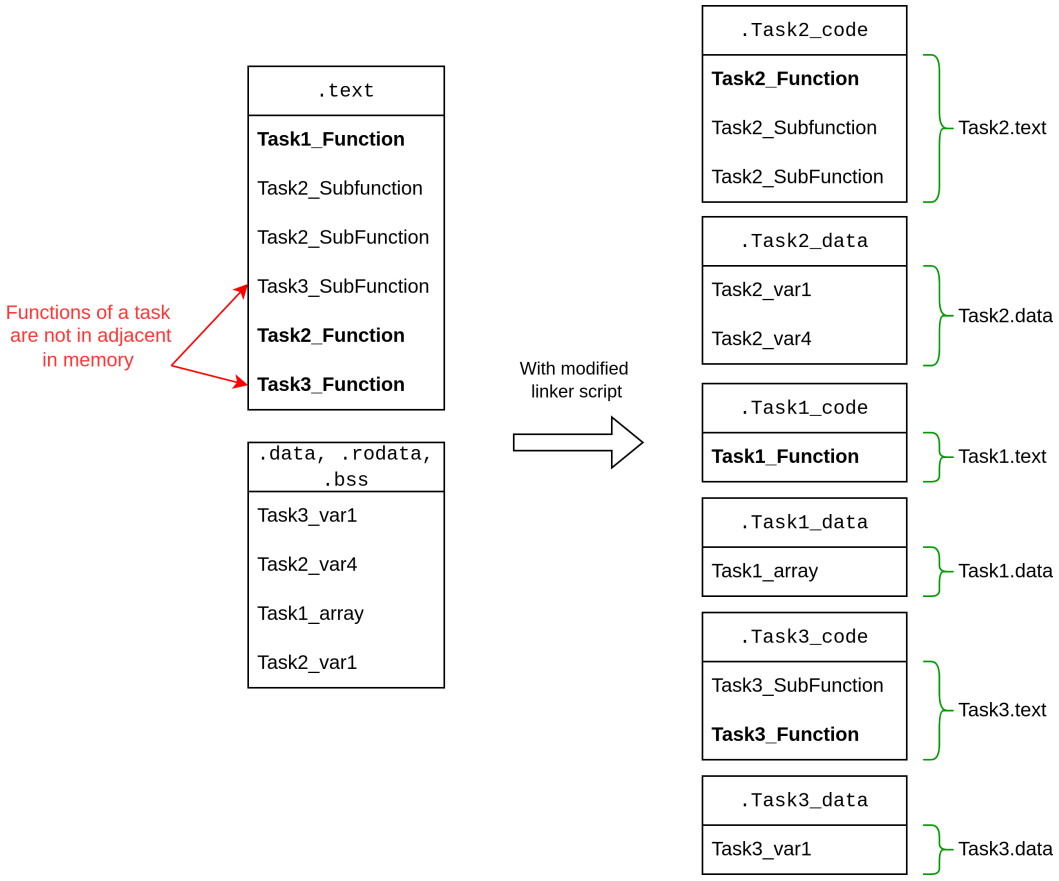


Figure 5.3: Code and data that belong to a task are grouped together by the linker. This is achieved by adding compiler attributes to the functions and data structures in the C-code.

5.2.3. Caches

The system incorporates two caches, one for the instruction fetches from the instruction fetch stage, and one for the memory read and write access managed by the Load-Store-Unit (LSU). Both caches

are identical and have access to the whole external memory space. The caches are implemented in Verilog and can be parametrized. Table 5.2 shows the cache configuration for each cache used for the implementation. The cache size is intentionally kept small to manage the implementation times for the FPGA design, which tend to escalate with larger cache sizes. The cache design was taken from a previous master thesis project [21] because it is compatible with the AHB interconnect with the Encryption and Authentication module that was developed by the same thesis and is also used in this design. The latter unit encrypts and authenticates data in blocks of 128 bits. Consequently, the cache line and the external memory are designed to operate with 128 bit cache lines. However, the cache design only supports a 16-way associativity, which can be considered disproportionate when compared to the cache size. Under typical circumstances, the cache would normally be larger, and the associativity would probably be smaller for an actual real-world application to reduce the cache complexity.

Table 5.2: Cache configuration for each of the two caches.

Parameter	Value
Cache size	512 byte
Cache line size	128 bit
Associativity	16-way
Replacement Policy	Round-robin
Write Policy	Write-back

5.3. Secure Boot

The secure boot for this platform is implemented in software and protects all code that runs on M-mode, i. e. has full privileges. This includes the the executable code and the data of the FreeRTOS kernel and the start-up code that runs before it. The user tasks are not included because their protection is optional, so time is saved during the start-up, which can be significant as cryptographic algorithms in software are comparatively slow on embedded systems. The secure boot uses a signature verification scheme instead of a simple hash-function to make it possible to update the program code. The secure boot follows the procedure from subsection 3.3.4.

The implementation shall use strong cryptography, leveraging mature and widely recognized algorithms to ensure high security. Keystone [20] implements a secure boot that is publicly available on Github [89]. The same algorithm was also used in its predecessor Sanctum, suggesting its reliability and robustness. The implementation is presented in Listing 5.1, and in detail, it performs following steps:

1. Calculate a 256-bit hash digest of the address range using SHA-3
2. Load public key from external memory and signature from a secure storage
3. Verify the signature against the hash using Ed25519
4. If signature verification is successful, continue start-up code. If it fails, enter endless loop

There are several aspects of the implementation that are crucial for ensuring security. First, the secure boot verifies the integrity of the external memory but it cannot protect itself. Therefore, the code of the secure boot must be put into secure memory. The simplest solution is to put it into OTP-memory that can be locked permanently. Also, it must be ensured that this code is the first thing that is run by the CPU, so the boot address must not be modifiable by an unauthorized attacker. In the implementation, this is achieved by adding the compiler attribute to the function, that it shall be put into the section “secure_bootloader”. This section must end-up in some secure memory when deployed a real SoC. Second, the start and end address of the memory must be specified in the linker script. In this implementation, they are declared as external variables because the function itself cannot obtain these addresses by itself. Lastly, the assumption for this secure boot to work is that the public key cannot maliciously be altered. The key is not confidential, as the confidential private part of the key pair is used during signing and not verifying. However, if the key can be replaced, an attacker could insert a malicious key and sign software with the corresponding private key. Then, it would be possible for an attacker to boot malicious software. Thus, the public key should be stored in secure memory as the

secure boot code as well. In the following, the secure boot and the key material will be treated in a similar way.

Listing 5.1: Implemenation of signature-based secure boot. The cryptographic functions are taken from Keystone's secure boot implemenation [89].

```
__attribute__((section(".secure_bootloader")))
void secure_bootloader()
{
  // Addresses are provided during linking stage
  extern uintptr_t __secure_boot_start_address__;
  extern uintptr_t __secure_boot_end_address__;

  // Hash over startup-code and kernel
  sha3_ctx_t hash_ctx;
  sha3_init(&hash_ctx, 64);
  size_t code_size = (size_t) ((void*)&__secure_boot_end_address__ -
      (void*)&__secure_boot_start_address__);
  sha3_update(&hash_ctx, (void*)&__small_secure_boot_start_address__, 1);
  sha3_final(FreeRTOS_kernel_hash, &hash_ctx);

  // Load public key and signature
  #include "use_test_keys.h"

  // Check signature
  int secure_boot_success = ed25519_verify(secure_boot_signature,
    FreeRTOS_kernel_hash, 64, secure_boot_public_key);

  // Verification failed
  if (!secure_boot_success)
    while (1) {}

  return;
}
```

5.4. FreeRTOS Task Isolation with PMP

This section explains how PMP is used to securely isolated FreeRTOS tasks from each other. To give a comprehensive explanation, this section is split in multiple subsections because of the interdependent nature of this method with various platform components.

The goal is to restrict a task's memory accesses to only its own executable code and its data, plus a set of services offered by the kernel. Optionally, additional regions can be specific for shared memory with other tasks, or memory-mapped peripherals. For the kernel services, also called system calls, an unprivileged section is introduced that contains functions that the user tasks are allowed to call. These functions will initiate kernel intervention to perform the requested operation on behalf of the tasks. This approach ensures that tasks do not directly use privileged functions, a mechanism that will be explained in detail in subsection 5.4.4.

For a task to be fully protected according to this description, at minimum 7 PMP-configs are needed, as shown in Table 5.3. Two are used for the unprivileged section, including system calls and standard library functions, because the TOR addressing mode is chosen (see subsection 2.2.2). It is followed by a region that only protects a 32-bit read-only variable which tracks whether the system is in privileged mode or not. This variable is required during system calls to decide whether a privilege switch to M-mode is needed. This variable should only be modifiable by the kernel so it is set read-only. Only one region is needed because being a 32-bit region, it is aligned to a power of 2 and can be described with the NA4 addressing mode. These three are fixed for all tasks and do not change during the whole runtime of a program. Then, there are two task-specific ranges, the code and the data, that change with each task switch to allow access only to the memory of the running task. Four regions are used, due to the TOR addressing mode, because the code and data can be of arbitrary size and are not necessarily a power of 2. Optionally and if available, further task-specific regions can be configured

with the residual regions. Table 5.3 presents the regions in detail which are also further explained in the following. Unfortunately due to the uneven number of used regions, one region cannot be used because the implementation only supports TOR regions until now.

Table 5.3: Mapping of PMP regions to address ranges in FreeRTOS implementation.

PMP region index	Addr. mode	Permissions	Label
Task specific: Changes with each task switch			
nRegions-1	-	-	unused
nRegions-3, nRegions-2	NA4 & TOR	rw	Optional task specific region
...	...	...	...
9, 10	NA4 & TOR	rw	Optional task specific region
7, 8	NA4 & TOR	rw	Optional task specific region
6	TOR	x	Task code end address
5	NA4	x	Task code begin address
4	TOR	rw	Task stack end address
3	NA4	rw	Task stack begin address
Fixed: Only configured once at start-up			
2	NA4	r	`privilege_status`
1	TOR	x	`__unprivileged_section_end__`
0	NA4	x	`__unprivileged_section_start__`

5.4.1. Assigning PMP-Regions to Task

At the moment when a task is created, the task's code and data regions, as well as the additional regions, are stored in the task's Task Control Block (TCB). In FreeRTOS, the TCB is an important data structure that holds all the information about a task, including its current state, stack pointer, priority, etc. The TCB is used by the scheduler to manage task switching, ensuring each task receives the appropriate amount of processor time according to its priority and state. It is stored on the heap and the task itself has no access to it. As it already holds other information that are needed during task switches, it is the optimal place to also hold the configuration of the PMP-regions of the task. The data structure definition of the TCB is shown in the C-snippet in Listing 5.2. It is extended with the field `xMPUSettings` if the PMP mode is enabled which is controlled by the macro `portUSING_MPU_WRAPPERS` in FreeRTOSConfig.h. This code is inspired by the ARM-MPU version [16], that is why the names of the fields and variables follow the naming conventions used there, including the frequent use of the prefix MPU.

Listing 5.2: Excerpt of the C-struct definition for the task control block (TCB) in the file tasks.c.

```
typedef struct tskTaskControlBlock
{
    volatile StackType_t *pxTopOfStack; /* Points to the top of task's stack */

    #if ( portUSING_MPU_WRAPPERS == 1 )
        xMPU_SETTINGS xMPUSettings;  /* Here the PMP regions are stored */
    #endif

    \* ...more fields... *\

    UBaseType_t uxPriority; /* The priority of the task.  0 is the lowest
        priority. */
    StackType_t *pxStack; /* Points to the start of the stack. */
    char pcTaskName[ configMAX_TASK_NAME_LEN ]; /* Name of the task */

    \* ...more fields... *\

} tskTCB;
```

Further, the data structure that holds a task's regions is presented in Figure 5.4. The first fields hold the 8-bit value of the PMP configurations as described in an earlier chapter in Figure 2.7. Because one 32-bit PMP-`config` hardware register holds the values of four 8-bit configurations, the field only holds a 32-bit value, where the region's 8-bit value is put at the position where the value will eventually end up in the hardware register. The next field is a mask that masks exactly these 8 bits with a one and the rest with zeros. This helps when the values are actually written to the PMP-`config` register to mask and modify only specific pmpconfig values in the register. The last field is straightforward, because it simply holds the address value of the region.

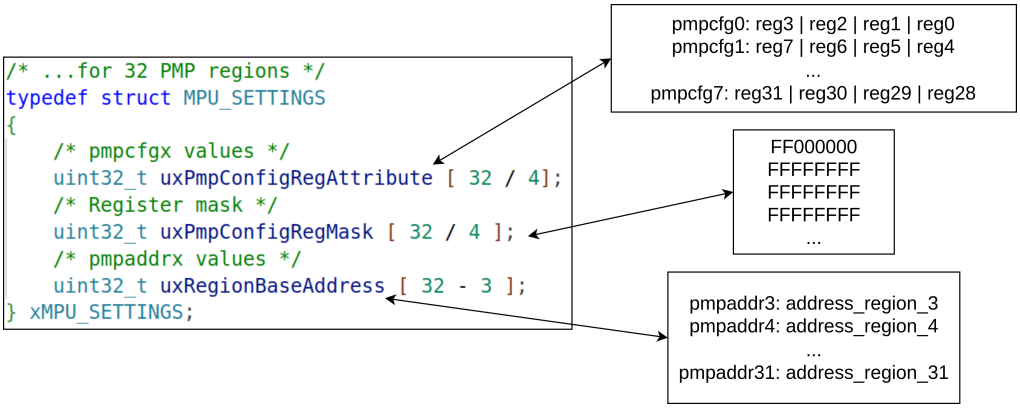


Figure 5.4: C-struct definition of xMPU_SETTINGS in the file portmacro.h. This example illustrates the layout of this struct for 32 PMP regions.

As explained before, these values are written for each task during the task creation. The next section explains how the task switch makes use of these structures.

5.4.2. Task Context Switch

During a SysTick interrupt, the context of the interrupted task is saved, and the tick counter is increased. FreeRTOS holds the counter value `xNextTaskUnblockTime` that stores the tick counter value at which the next task is becoming ready again. If the new tick counter value matches this value, a task switch is initiated. A task switch can also be triggered in different ways, for example, when a task puts itself into the blocked state. Then, the kernel takes over and initiates a task switch.

When this happens, the assembly code in Listing 5.3 is executed. It starts with the actual task switch in the function `vTaskSwitchContext`, in which the next task to be run is determined. FreeRTOS maintains a list of ready tasks, organized by priority. In this, it selects the highest priority task that is ready, and writes the address to this task's TCB to `pxCurrentTCB`. After that, this value is used to read the TCB's fist entry, the task's stack pointer register `sp`, which is loaded to the CPU's `sp` register. Finally, the call to `vPortPmpSwitch` has been added that reads the PMP-settings from next task's TCB, and applies them to the PMP hardware registers.

Listing 5.3: Assembly code of context switch that is run right before the next task is activated.

```
1 switch_context:
2     jal vTaskSwitchContext  /* Perform task switch / context switch */
3     load_x s0, pxCurrentTCB /* Load TCB of next task */
4     load_x  sp, 0( s0 )     /* Read sp from first TCB member */
5     jal vPortPmpSwitch      /* Write PMP regions from TCB to the HW-registers */
```

The function `vPortPMPSwitch` is implemented in assembly language because it mostly interacts with the PMP registers of the CPU directly. The function is outlined in Figure 5.5. At the beginning, all dynamic PMP registers are turned off by resetting the address mode values of all dynamic configurations to 00. The first three static regions remain untouched. The next step writes all the addresses from the tasks `MPU_Settings` to the PMP address registers (`pmp_addrx`). After that, the dynamic PMP configurations are reset using the masks, so they can be set in the subsequent final step. The masking is important here to preserve the first three static configurations, because of the particularity of RISC-V that one

PMP configuration register holds four actual configurations. The first step of turning all configurations off might seem unnecessary because the configurations will be overwritten in a following step anyway. However, this is done to avoid having an undefined state in the second step when the addresses are written.

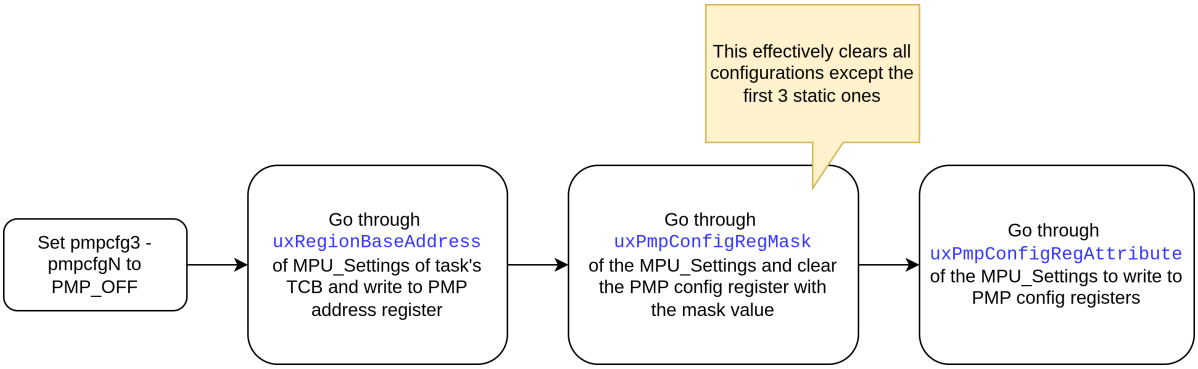


Figure 5.5: Updating PMP configuration during task switch as implemented in vPortPMPSwitch.

5.4.3. Trap Handling

The trap handler is a central piece of software for this FreeRTOS implementation. It handles exceptions and interrupts that occur during the runtime of the program. These can be triggered from the outside, for example, the timer interrupt for the `SysTick` of FreeRTOS, but they can also happen from within the CPU, like an illegal instruction, an access fault, or an environment call. The timer and environment call interrupt are explicit requests that are required for the functionality of the FreeRTOS kernel. The others are fault interrupts and normally are not intended by the programmer to happen and occur when some part of the system is misbehaving.

The trap handler in the implementation for this thesis has the following tasks:

- Increase the `SysTick` and possibly initiate a task switch when a timer interrupt occurred
- Manage system calls that are registered as environment calls or `ecalls`
- Enter an endless loop in case a fault occurred (fault-specific fault handling can be added if needed)

The behavior of the trap handler for these three tasks is illustrated in the activity diagram in Figure 5.6. The green blocks represent the interrupts (system calls and systick) necessary for the proper functioning of the FreeRTOS kernel. These are followed by an action (blue blocks). The red block represents the fault exceptions. The yellow block represents a verification step that increases the security for systems calls. When a trap happens, the RISC-V core writes the reason for it to the `mcause` register. The trap handler uses this value to decide which action to take. It also indicates whether it is an internal exception or an external interrupt by setting the Most Significant Bit (MSB) in the `mcause` register.

If the cause is a timer interrupt, the trap handler will increment the SysTick counter, which is an essential input for the scheduler. The scheduler tracks the status of all tasks and based on the new value of this SysTick counter, checks whether a task has become ready that has a higher priority than the currently running task. However, at the beginning of the interrupt, the CPU's registers hold the values that are needed by the interrupted task once the task is activated again. Therefore, it is important that the trap handler first saves the current context by copying all register value onto the task's stack. After that, the stack pointer is changed to point to a special interrupt stack, so the task's stack is not used for the following steps. At this point, the trap handler is able to execute code without corrupting the status of the task. Now, the trap handler increments the SysTick and the scheduler is initiated to check whether a task switch is needed. If this is the case, then the assembly routine (Listing 5.3) from the previous section is executed. In the last step of this interrupt, the context of the current task is restored by copying the context from the task's stack back into the hardware registers. This task can either be the task that was interrupted by the trap handler, or the task that got activated by the context switch. Finally, the trap handler is left and the task keeps running.

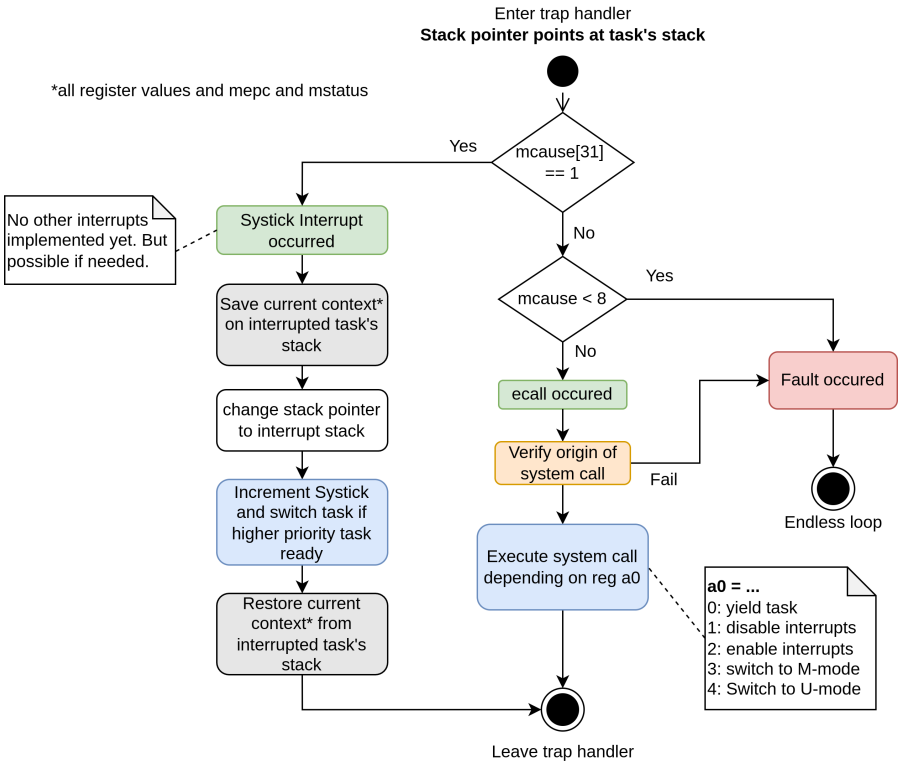


Figure 5.6: Execution flow of trap handler.

If the cause is an `ecall`, the trap handler is requested to perform a system call. There are multiple system calls available. Which one of them is called is specified by the value in the register a0 that is present at the moment of the system call when the trap handler is entered. That means, that the piece of the software that triggers a system call by executing the instruction `ecall` has to write the system call value to the a0-register before that. Before a system call is executed, the trap handler checks whether the system call was initiated from the system call address space to hinder user tasks from initiating themselves but use the provided calls of the FreeRTOS implementation. This is an added security feature and will be explained in more detail in a following section. Then, trap handler reads the value of a0 and jumps to the according function. System calls 1 to 4 are comparatively small operations because they only write to a few hardware registers without the need to save and restore the whole task's context. System call 0, yield task, is more complex because it requests the scheduler to be set to the blocked state and to let a different task run. For this reason, this system call saves and restores the whole task's context.

Lastly, the trap handler also handles faults that happened during the runtime. For this work, no advanced fault handling is implemented. Instead, all faults result in an endless loop, which effectively stops the execution of the FreeRTOS kernel. Illegal system call requests are also redirected here.

5.4.4. System Call Handling

System calls offer the user tasks to request services from the kernel that the tasks are not authorized to execute. This is important because it creates a boundary between the user space (unprivileged) and the kernel space (full privileges), which is a requirement for this design. When a system call is initiated, the kernel performs the operation on behalf of the user task. It only offers functionality that is reasonably required by tasks, such as a task delay, or reading and writing files. Privileged functions that user tasks should not use directly are not offered and can only be initiated by the kernel. Examples are disabling all interrupts, or reset the system.

The implementation of system calls in this thesis follows the wrapper-like technique similar to earlier MPU-FreeRTOS versions. This technique, as it is, is not secure because it can be exploited if an

attacker knows the system well and uses finds certain buffer-overflow weaknesses as explained in [81]. A more comprehensive secure system call handling framework for FreeRTOS for ARM platforms [14] was explored but later discarded because the software-engineering efforts proved to be too significant. The improvements of this framework are an Access Control List (ACL), which allows for task specific permissions management for kernel functions, and a complete re-implementation of each kernel service as its own system call - instead of keeping all the functions as they are and wrapping them with two simple system calls. Another difficulty is that the differences in the ARM and the RISC-V ISA makes a translation of the solution complex. For example, the system call handling in ARM relies on instructions that request the current privilege mode. In RISC-V, this functionality not implemented as a deliberate design choice.

Consequently, the simpler wrapper-like system call handling was retained to ensure timely project completion. To still make it adequately secure by avoiding exploits as described in [81], a simpler mechanism that verifies whether system calls are issued from within the address range of the system calls. This concept has already been mentioned in the previous sections and will explained in detail now. The system call handling in the following is explained by the function `vTaskDelay(int number_ticks)` as used in a previous chapter in Listing 2.1. This section will first explain the deficient wrapper-like system call method, explain how it can be exploited, and then present an improved design.

FreeRTOS-MPU's Wrapper-like System Call Handling

Whether the system call wrappers are used or not is defined by the value of `portUSING_MPU_WRAPPERS`. If it is 1, the wrapper-like system call mechanism is activated that wraps the original function with functions that temporarily raise the privilege mode for the execution of this function. This works in two steps. First, the original privileged functions are replaced by the MPU-functions which can be seen in Listing 5.4. This listing only shows a part of all functions but in total, all functions that require privileged intervention are replaced by an MPU version.

Listing 5.4: mpu_wrappers.c taken from the MPU-FreeRTOS for ARM platforms.

```
#if( portUSING_MPU_WRAPPERS == 1 )
    /* Map standard tasks.h API functions to the MPU equivalents. */
    #define xTaskCreate                         MPU_xTaskCreate
    #define xTaskCreateStatic                   MPU_xTaskCreateStatic
    #define xTaskCreateRestricted               MPU_xTaskCreateRestricted
    #define vTaskAllocateMPURegions             MPU_vTaskAllocateMPURegions
    #define vTaskDelete                         MPU_vTaskDelete
    #define vTaskDelay                          MPU_vTaskDelay
    #define vTaskDelayUntil                     MPU_vTaskDelayUntil
    #define xTaskAbortDelay                     MPU_xTaskAbortDelay
    /* ... etc. ... */
#endif /* portUSING_MPU_WRAPPERS */
```

Second, the according MPU function wraps the original function with the functions `xPortRaisePrivilege()` and `vPortResetPrivilege()`. This is shown in Listing 5.5 where the function `vTaskDelay()` resides between these two function calls. That effectively means that privileges are temporarily raised by switching the CPU to M-Mode during the execution of the original function, and switching back to U-Mode after completion. If the system is already in M-mode, the two wrapper-functions do not switch and stay in M-Mode during and after the system call. The variable, `xRunningPrivileged` informs the reset privilege function about whether or not privileges have to be reset to U-Mode. The wrapper functions implement an environment call with the respective number. Upon an environment call, the trap handler from Figure 5.6 is started that with `mcause` being 8. a0 holds either the value 3 for switching to M-Mode, or 4 to reset back to U-mode.

Listing 5.5: Example wrapper function for vTaskDelay.

```
#if ( INCLUDE_vTaskDelay == 1 )
    void MPU_vTaskDelay( TickType_t xTicksToDelay ) /* FREERTOS_SYSTEM_CALL */
    {
        BaseType_t xRunningPrivileged = xPortRaisePrivilege();
        vTaskDelay( xTicksToDelay );
```

```
6           vPortResetPrivilege( xRunningPrivileged );
7       }
8   #endif
```

Attacks on this Method

The security flaw explained in [81] exploits the fact that the privileges can be raised to M-Mode with an environment call. This design relies on the assumption that only the wrapper functions use this environment call but there is nothing stopping a malicious application to call it. This is illustrated as *Attack 1* in Figure 5.7. If this attack is executed, the trap handler will switch to M-Mode and jump back to the address after the ecall in the task code. Then, the task has higher privileges and has control over the whole system. Alternatively, the attacker could call the function `xPortRaisePrivilege` that then executes the environment call on behalf of the task. The same could be done with the subfunction `portRAISE_PRIVILEGE`. The function identifier will not be provided to the task code, therefore, the address must be called directly. This function address can be reversed-engineered without too much effort. Either the function call is programmed into the task code already, which is conceivable for malicious third party code, or a buffer-overflow attack can be used to redirect the control flow and call the function, as described in [81].

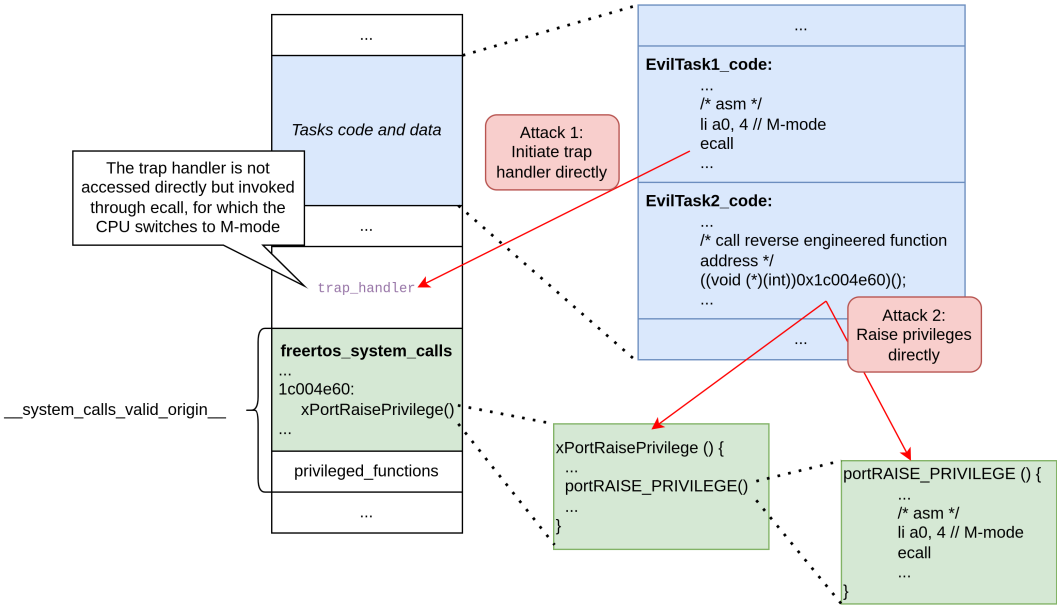


Figure 5.7: Possible attacks on the unmodified wrapper-like system call structure.

If these attacks are possible, the attacker can take full control over the system and can deactivate the PMP rules. Consequently, the security measures described in this thesis would be rendered ineffective. In the following, two measures preventing these exploits will be explained.

Hardening Wrapper-like System Calls

The approach to prevent these attacks is to verify whether a system call was initiated from the region `__systems_calls_valid_origin__`. This region is added to the linker script and includes the system calls section that is accessible to the user tasks, as well as the privileged functions of the FreeRTOS kernel. It shall only be allowed to initiate system calls from this range. Because there are two entry points for this attack, two checks have been added.

Attack 1 is prevented by verifying the `mepc` in the trap handler. When an environment call happens, this register is loaded with the address of the instruction following the environment call. This is controlled by the CPU hardware, thus the attacker cannot tamper this value. In the trap handler, an additional check is added that compares the `mepc` to the boundaries of the system call region, as shown in Listing 5.6, lines 7 to 12. This check was already included in Figure 5.6 in a previous section.

Listing 5.6: Added verification of environment call origin in trap handler in file portASM.S.

```
environment_switch:

    # SysCall Security: Check whether origin of call is from system calls
    csrr    t0, mepc

    # Load the start address into t1 and the end address into t2
    la      t1, __system_calls_valid_origin_start__
    la      t2, __system_calls_valid_origin_end__

    # Verify that mepc is in that range, otherwise throw exception
    blt     t0, t1, is_exception
    bgt     t0, t2, is_exception

    # Jump to the call depending on the value in a0
    li t0, 4
    bgt a0, t0, ecall_end
    la t0, 1f
    slli a0, a0, 2
    add t0, t0, a0
    jr t0
1:
    jal x0, ecall_yield
    jal x0, ecall_disable_interrupt
    jal x0, ecall_enable_interrupt
    jal x0, ecall_switch_to_machine
    jal x0, ecall_switch_to_user
```

Attack 2 is prevented in a similar way by adding a check to the system call. A difficulty is the subfunction `portRAISE_PRIVILEGE` because it gives the attacker an additional access point for the attack. FreeRTOS wants to separate port-specific code from general code that runs on all platforms. This is the reason why this subfunction is introduced here. However, for securing the system calls, the first step is to flatten the hierarchy such that the `ecall` is executed from the `xPortRaisePrivilege()`. With this, it is possible to verify whether the function has been called from the system calls range. This check is added as presented in Listing 5.7. The difference in this case is that the check happens *after* the `ecall` in line 15 and the privilege status is updated. The address range check follows in lines 22 to 30. The reason why it is verified after the switch is that if it is checked before, and attacker could simple call the address of the ecall instructions and would skip the check. If it is placed after, the check is guarenteed to be executed because the processor will continue executing the following code. If the check fails, the program will throw an exception and be stuck in an infinite loop.

Listing 5.7: Added verification of environment call origin in function xPortRaisePrivilege in file mpu_wrapper.c.

```
BaseType_t xPortRaisePrivilege( void )
{
        /* Check whether the processor is already privileged. */
        BaseType_t xRunningPrivileged;
        xRunningPrivileged = portIS_PRIVILEGED();

        /* If the processor is not already privileged, raise privilege. */
        if( xRunningPrivileged != pdTRUE )
        {
            __asm__ __volatile__ (
            "   .extern privilege_status    \n"
            "   li  a0, 3                   \n"
            "   ecall                       \n"
            "   # Update privilege_status   \n"
            "   la  a0, privilege_status    \n"
            "   li  t0, 3                   \n"
            "   sw  t0, 0(a0)               \n"
            "                               \n"
```

```
19              "   # Check whether origin of call is from system calls \n"
20              "   mv  t0, ra      \n"
21              "                   \n"
22              "   # Load the start address into t1 and the end address into t2 \n"
23              "   la  t1, __system_calls_valid_origin_start__   \n"
24              "   la  t2, __system_calls_valid_origin_end__     \n"
25              "   .extern is_exception        \n"
26              "   blt t0, t1, is_exception    \n"
27              "   bgt t0, t2, is_exception    \n"
28          ::: }
29
30          return xRunningPrivileged;
31  }
```

This method improves the security of the system calls by preventing simple attacks as presented in this section. However, this solution is still not perfectly secure because an attacker could still outsmart these checks with intelligently crafted the stack using a BOF attack. The stack would need to be crafted in a way that it contains a return address that points to a return address in the system calls memory, that again jumps back to the evil tasks from Figure 5.7. A way to prevent this could be, to not check only whether the return address is in the valid range of `xPortRaisePrivilege`, but to restrict it to only a set of allowed return addresses inside the system call address range. Investigating and improving the security of this solution can be subject of future work. For now, this implementation can be considered hardening of the system.

5.4.5. Dynamic Memory Allocation Support

Dynamic memory allocation is an inherently non-deterministic operation and and must be treated with caution when used for real-time embedded systems. This is because the memory allocator is often provided by the operating system, which can differ from platform to platform. Also, the fragmentation of the heap memory results in unpredictable allocation times and potential allocation failures. For these reasons, FreeRTOS ships their own memory allocators with FreeRTOS. Memory allocation schemes can come with different levels of complexity, therefore, FreeRTOS decided to offer 5 different options. The memory is allocated in the `.heap` memory section and FreeRTOS contains the five allocator files heap_1.c, heap_2.c, heap_3.c, heap_4.c, and heap_5.c. The programmer can chose an allocator by including the file in the sources for the compilation process.

All of these files implement at least the two functions `void *pvPortMalloc(size_t xWantedSize)` and `void vPortFree(void *pv)`, which resemble the standard C functions `malloc` and `free`. These two functions implement the operation of allocating and deallocating memory, meaning they handle the process of reserving a specific amount of memory and then later releasing it when it is no longer needed. heap_1.c implements the simplest allocator. It allows only allocation but no deallocation of memory. This avoids memory allocation issues, such as determinism and fragmentation. It implements the function `vPortFree` for compatibility, but it throws an error when being called. heap_2.c also allows the deallocation memory by using a simple best-fit algorithm that uses the free block of memory closest to the requested size. However, this allocator is superseded by heap_4.c and is not recommended for use anymore. heap_3.c simply provides a wrapper around the standard C library's `malloc` and `free` functions. This means that the library's own memory management functions are used. This can be useful for systems that already have a reliable standard library implementation. heap_4.c and heap_5.c both implement more advanced memory allocators to improve efficiency and flexibility. heap_4.c uses a best-fit algorithm, the same as heap_2.c, but it improves it by coalescing memory blocks during deallocation. This reduces the effect of fragmentation. heap_5.c builds on this by adding support for multiple memory regions, which allows memory blocks to be allocated in different memory banks. This can be useful for system with more complex memory architectures. This and more detailed explanations can be found in [30].

For this work, PMP-support for memory allocation has been added that works independent of these different memory allocators. Therefore, all allocators are supported. The solution is shown in Listing 5.8 and is similar to the wrapper-like system calls. The memory allocation must be handled by the kernel, thus it requires a system call. The wrapper-function `MPU_pvPmpMalloc` calls the memory allocator func-

tion of one of the heap_x.c files. Before this call, the privileges are raised in line 4, and end the end of the function, the privileges are reset, similar to all the other system calls. Additionally, the function `xAddMallocPMP()` is added, which implements the PMP-support.

Listing 5.8: Added function for updating MPU_SETTINGS of task to ensure PMP access rights to the newly allocated memory for the task.

```
1  void *MPU_pvPmpMalloc( size_t xSize )
2  {
3      void *pvReturn;
4      BaseType_t xRunningPrivileged = xPortRaisePrivilege();
5      pvReturn = pvPortMalloc( xSize );
6
7      /* Added for PMP support */
8      xAddMallocPMP(pvReturn, xSize);
9
10     vPortResetPrivilege( xRunningPrivileged );
11     return pvReturn;
12 }
```

This added function gets the start address and size of the newly allocated memory region and goes through the PMP configurations that are present in the current PMP hardware registers. It scans it for unused regions by starting at the bottom and going up. It stops at the first configuration that is disabled, and then, writes the start address to the first free PMP-register. After that, it writes the end address (start + size) of the allocated memory to the following free register. The address range is addressed with the TOR mode to make arbitrary sizes possible, and only read and write accessed are permitted. This configuration is also added to the task's MPU_SETTINGS so it is also activated at the next task switch. The example in Figure 5.8 demonstrates this by allocating 100 bytes of memory on the heap.

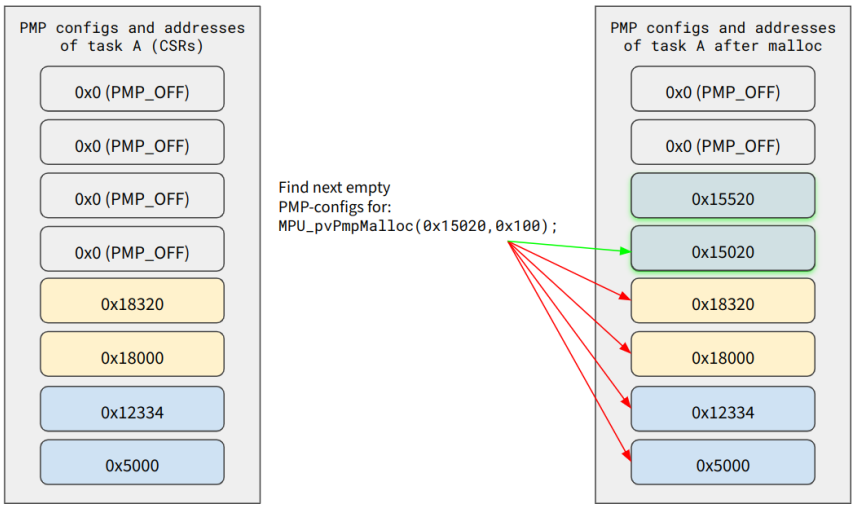


Figure 5.8: Scanning the PMP registers for unused regions. The algorithm starts at the bottom configuration and goes upwards until it finds a region that is turned off.

The procedure for the the `MPU_pvPmpFree()` is identical to the `MPU_vPmpMalloc` with the difference that `xRemoveFreePMP(pv)` is used to remove the regions from the task's PMP config in the TCB. This function searches the region with the address of the pointer address that is provided to the free-function. Once it is found, it gets deactivated in the TCB and the new region configuration is written to the hardware registers to ensure the effect takes place immediately.

5.5. FreeRTOS Task Encryption

The task encryption and authentication should optionally be enabled for specific tasks. When choosing cryptographically strong algorithm, this operation requires significant computational power, However,

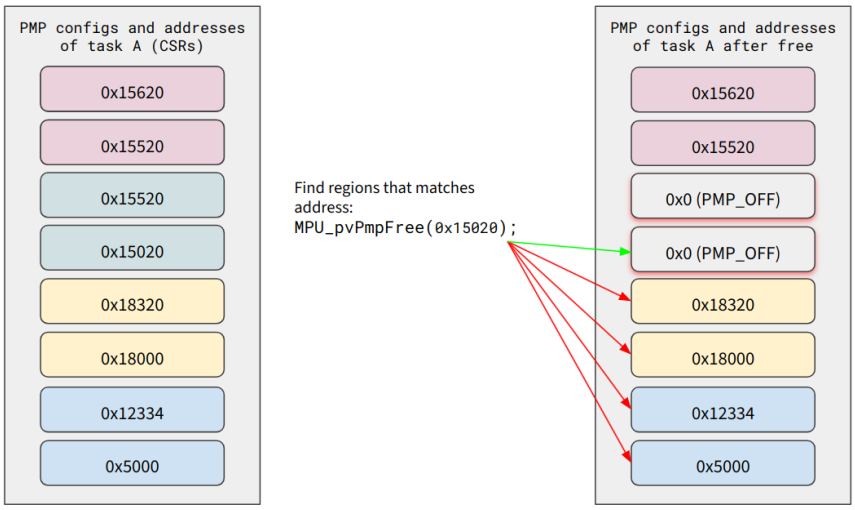


Figure 5.9: Scanning configurations, bottom to top, for address to be free'ed. After the address has been found, the regions are turned off and removed from the MPU_SETTINGS.

many embedded systems have limited processing capabilities which means that the encryption and authentication operation would induce a high overhead in memory latency and energy consumption. For this thesis, only the encryption of data is implemented. The authentication task was excluded because it would require significant time and effort and its contribution of new knowledge is limited, given its similarity to the encryption module.

As illustrated in Figure 4.1, data that is stored on the external memory is vulnerable to modifications and eavesdropping, thus, there the data has to be secured. Inside the SoC the data is secure because the attacker cannot easily access the internal memory. Therefore, the data has to be encrypted and authenticated when it leaves the internal memory to the external memory. Similarly, it has to be decrypted and verified when it enters the internal system again. This would prevent an attacker from modifying critical software and reading sensitive information.

This encryption and authentication unit has to be put into the SoC, where it is inaccessible to the attacker except for the signals that leave the unit to the external memory. Otherwise, the attacker could read the plaintext traffic between the processor and the unit. Consequently, the data in the cache is not secured and contains the plaintext. Only in case the cache writes back memory, the data is encrypted and authenticated before being stored in the external memory. Respectively, if the processor needs data that is not available in the cache (cache miss), the cache requests encrypted data from the external memory. This data is decrypted and verified by the unit before making it available to the cache. To reduce the execution time, this unit shall be implemented as a hardware accelerator. [21] developed the Embedded Memory Security module which effectively is a encryption and authentication module that secures the external memory. The work analyzed possible attacks on external memory modules for which this module was proposed as a countermeasure. Different hardware accelerated algorithms for encryption and authentication were compared in terms of throughput, area, and energy consumption. The selected algorithm for encryption is Prince [90], which is a lightweight block cipher that outperforms other algorithms because of its remarkably low latency and area. For the authentication, SipHash [91] was chosen for similar reasons.

5.5.1. Combining Encryption and PMP

The encryption and authentication feature shall be applied to selected tasks' code and data. The encryption unit shall be controlled by the CPU hardware and not by software. If it was controlled by software, the software has to be aware that a cache miss happened and implement some code that sends the data to the encryption unit for external memory data traffic. In the proposed design, the software shall not have to deal with the memory handling, rather the memory side should be transparent to the software. The software should read and write to the external memory and the cache and memory

encryption unit shall not rely on any direct intervention of the software that uses it.

All tasks are already protected by the PMP encapsulation, for which the transparency requirement from above is true: During a task switch, the PMP settings are configured to restrict the memory access for the next running task. For this, the configuration is written to hardware registers and some logic in the CPU monitors the task's memory accesses and blocks illegal actions. This is transparent for the programmer because the details about how the memory protection works, is abstracted away for the software. Because of the similarities, this design expands the PMP configuration to also contain the encryption and authentication settings. The PMP configuration register format, which is defined by the RISC-V Instruction Set Manual [35] and described already in subsection 2.2.2, contains two reserved bits. The bits at position 5 and 6 are set to zero and not used. These two reserved bits are now used to enable the encryption and authentication unit. By that, the encryption configuration for each task is integrated into the tasks' PMP settings to facilitate the implementation of the design. This requires additional hardware support.

In the core of the design, two PMP units exist. One unit for the instruction fetch stage and the other for the LSU. Figure 5.10 shows a simplified model of the CPU that shows the relevant units involved. The added signals are highlighted in pink and green. Using the encryption unit for the prefetcher is straightforward. When an instruction is requested, the cache will be interrogated first. In case of a cache-hit, the cache holds the data for the requested address already and can provide it. In case of a cache-miss, the cache requests the cache line, containing the requested data, from the external memory. It does not access the external memory directly because the encryption and authentication unit is placed in between. This unit was modified to have an *enable*-input that activates it. If the input is 0, the unit does not perform any operation on the data and forwards it as it is. The PMP-unit of the prefetcher indicates whether the external memory of that region is encrypted or not. This depends on the values of the unused bits in the PMP-register that are now used for this encryption and authentication feature. If it is, the cache requests the data from the external memory through the encryption unit and enables it. That way, the read data gets decrypted and is sent to the instruction-cache that forwards it to the prefetcher. In case the region is not encrypted, the encryption unit does not get enabled and it delivers the external memory directly to the cache and prefetcher without modifying it.

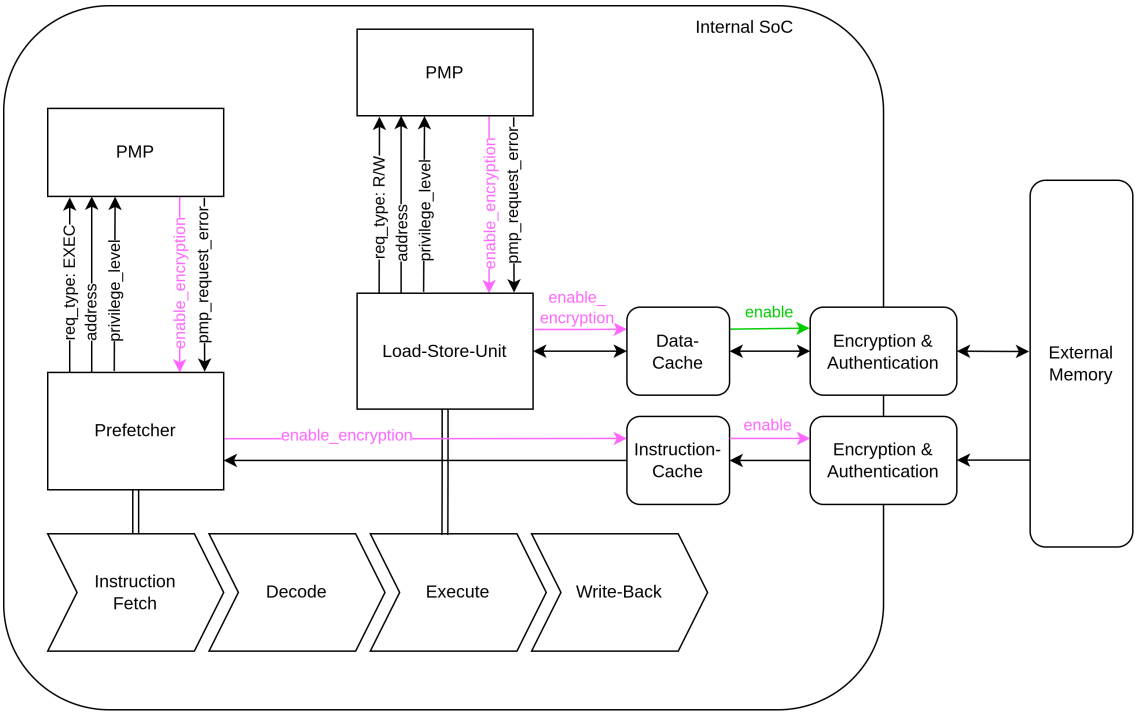


Figure 5.10: Integration of Encryption & and Authentication module into SoC.

For the LSU, the situation is more complex. The read operations function identically to those of the prefetcher. However, the write operations present a unique challenge. Specifically, due to the write-

back cache type, data is not immediately written to the external memory. Instead, the write-back method temporarily stores modified data in the cache, deferring the write to external memory until necessary, when a cache line gets evicted by a more recent cache line. This is why the enable signal between data-cache and encryption unit in the previous figure is highlighted in a different color. Figure 5.11 shows how the cache has been modified to manage the delayed write-back encryption. The main addition to the cache design is to have an additional *enc* bit for each cache line, next to the valid and dirty bit. This bit indicates whether the data of the cache line has to be encrypted when it is written back to the external memory. It acts as a buffer to store the signal from the PMP-unit of the LSU together with the cache line. This signal is only used for write operations, and the signal directly from the LSU is used for read operations. This signal is multiplexed depending whether it is a read or write operation.

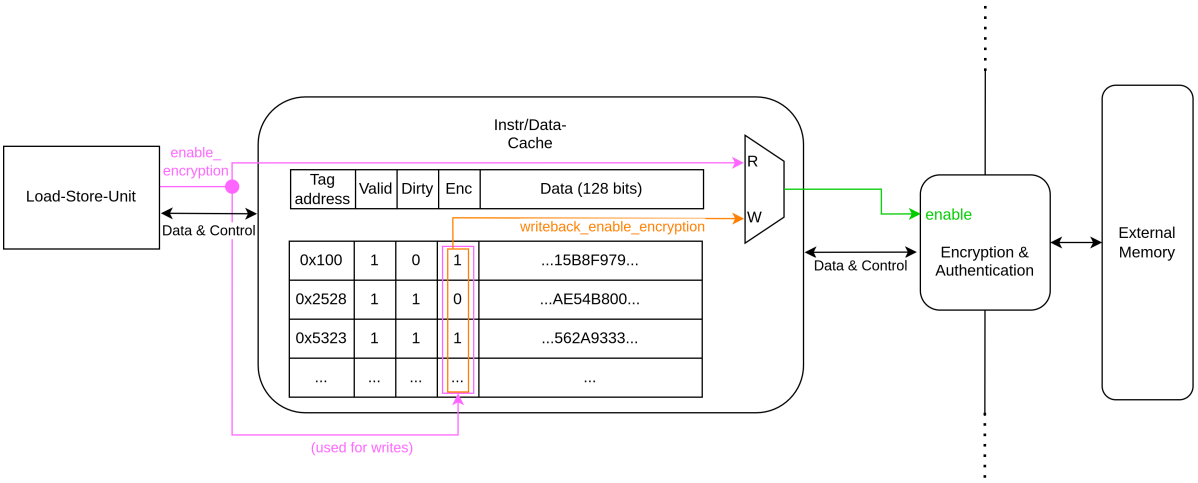
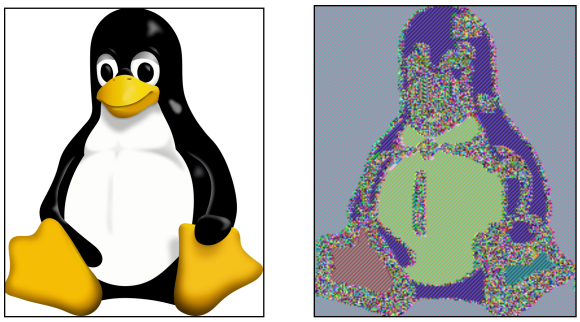


Figure 5.11: Modifications in cache to enable external memory encryption.

5.5.2. Address-Dependent Encryption

A major drawback of the block-wise memory encryption is that a block of data is always encrypted in the same way, such that an input always results in the same output. An attacker may not be able to figure out the plaintext directly, but it is possible to learn about the structure of the plaintext. This is illustrated in Figure 5.12, in which an image is encrypted pixel by pixel. While every pixel has a completely different color after encryption, all pixels with the same color are encrypted to the same output color. This weakness applies the same way to data in the memory. An attacker can learn about the structure of the application and even worse, if the attacker knows about the plaintext value that is stored at an address, all memory locations with the same encrypted value, hold the same plaintext. An attacker could also copy the ciphertext value to other locations in the memory. Encryption schemes that work like this are called deterministic because one input always leads to the same output [61].



Original image Encrypted in *Electronic Codebook* mode

Figure 5.12: Encryption in ECB mode of image reveals information about the structure [92].

The classical way to address the weakness is to chain the encryptions such that the previous operations

influence following encryptions. The plaintext blocks in Figure 5.13 include the ciphertext of the of the previous block, by XOR'ing it with the plaintext. Additionally, a random Initial Vector (IV) is provided for the first block. By using different IVs, even the encryption of the exact same blocks will lead to different ciphertexts. For the decryption, the IV and all the previous blocks in the right order must be available. Therefore, the IV is not secret and is sent to the receiver along the ciphertext. There exist different ways to combine the operations, which are also called *modes*. The figure here shows the Cipher Block Chaining (CBC) mode as an example. However, using this for the memory encryption presents challenges. Firstly, there is no obvious order in which the memory blocks can be chained. It could be chained based on the order of write operations, but then the chain would get destroyed as soon as a memory location is written twice. Also, the order would have to be tracked somewhere, which would result in additional hardware logic.

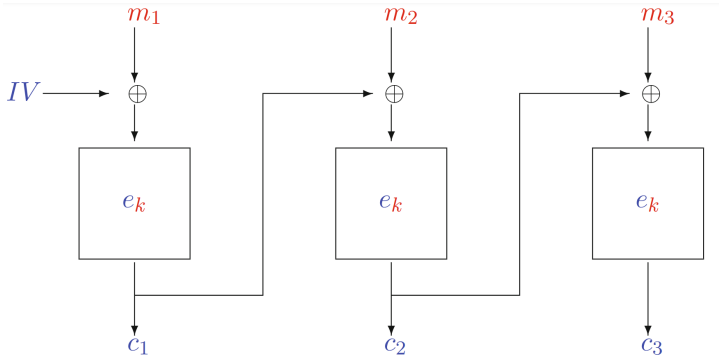


Figure 5.13: Encryption of multiple plaintext blocks *m* with a key *k* to ciphertexts *c* using the Cipher Block Chaining (CBC) mode. For the first block, a random Initial Vector (IV) is provided [61].

Tweakable Block Ciphers [93] are a suiting alternative for including non-determinism of a cipher while using the same encryption key. This cipher includes an additional input parameter, independent of the key and the plaintext, without introducing a dependency between different operations. This approach is used for the memory encryption of this thesis and the address of a memory cell is chosen to be the tweak. Before the data enters the encryption unit, it is XOR'ed with the memory address of the unit. However, using the memory address as a tweak is not perfect because for the same address the encryption stays deterministic. Nevertheless, the security is enhanced significantly considering the negligible overhead in design complexity. Figure 5.14 shows the encryption unit that combines the address (signals in orange) with the plaintext before encrypting. Likewise, the decrypted value must be XOR'ed again so the original plaintext value is restored.

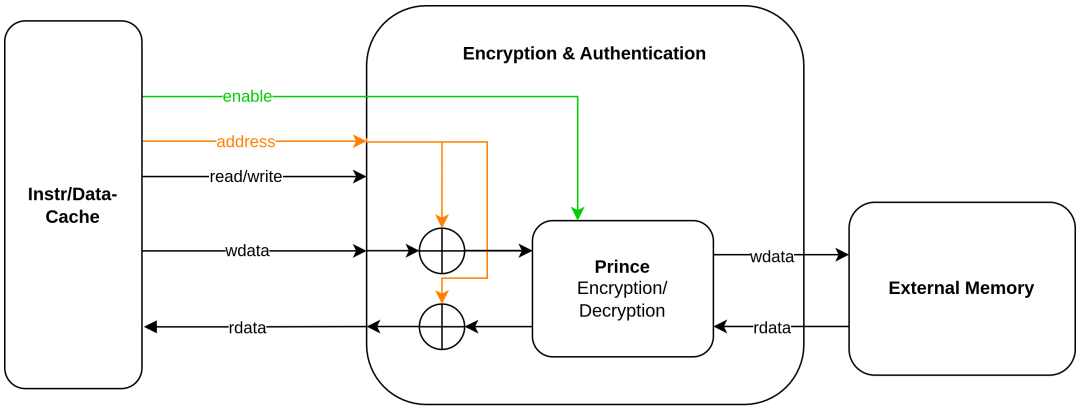


Figure 5.14: Modifications in encryption unit to include dependency to the address during encryption.

6

Validation & Results

This chapter demonstrates the added measures are implemented correctly and evaluates the performance of selected features. First, the experimental setup is explained to the reader in section 6.1. After that, test cases are described and conducted which demonstrate the correct workings of the implemented measures in section 6.2. Then, the impact on the real-time capability of the PMP integration into FreeRTOS is inspected in detail with different example test setups in section 6.3. Then, the performance of the encryption unit is evaluated when used for FreeRTOS tasks in section 6.4. As PMP is a hardware feature of RISC-V, the additional area of its use is reported in section 6.5. Lastly, the results and findings are discussed in section 6.6.

6.1. Experimental Setup

The security enhancements were developed in the Verilog description language, as specified in the previous chapter. For simulation and implementation of the design, Xilinx Vivado 2023.1.1 was used on an Ubuntu Linux machine, and the system configuration is summarized in Table 6.1. For measuring the performance, the design was implemented on a Xilinx PYNQ-Z1 board and was run with a CPU clock frequency of 20 MHz. The frequency is limited by the critical timing path in the cache access logic. In this design, the data in the cache is accessed in a single cycle, so improving this design could increase the frequency. One way could be to reduce the complexity of the cache design to shorten the critical path. Another way could be to pipeline the cache accesses and perform them in two cycles. The CV32E40S core would support this but the memory side including the caches have to be adapted. The memory access time to the BRAM typically is significantly longer than to the cache. Therefore, the BRAM access time was artificially delayed to take 50 times as many cycles as cache access.

Table 6.1: System configuration parameters.

Configuration flag	Value
CPU clock frequency	20,000,000
Cache memory access time	1 cycle
BRAM memory access time	50 cycles

6.2. Security Feature Validation

This section demonstrates the correct workings of the secure boot, the PMP task encapsulation, and the dynamic memory allocating with PMP support. For the demonstration, the secure boot is implemented and run on the FPGA, while the other two cases are run in simulation because they the simulation graphs offer valuable information about the system's behavior.

6.2.1. Secure boot

For the secure boot, the implementation of Keystone's secure boot was used. To verify that it works correctly, first, a successful secure boot is performed and the time is measured to evaluate the start-up latency, and second, the executable code is modified before start-up to show that it does not boot up.

For setting it up, a public key, an FreeRTOS application, and valid signature must be loaded to the system. This includes generating a key pair, hashing the start-up code and the FreeRTOS kernel, and calculating the signature. For this, the same C source as in the secure boot implementation was used and the public key and signature were stored in the *secure boot & key material* section, see Figure 5.2. This section should be put into protected memory when used in a real-world application, however, in this setup, it is stored in the BRAM alongside the code and data. The secure boot verifies the integrity of the sections privileged_data, privileged_functions, freertos_system_calls, text, data, rodata, and bss. The secure boot measured here starts at instruction address 0x1c000380 and ends at address 1c0523e4, resulting in a total of 42 Kbyte.

Table 6.2: Secure boot execution time over 42 Kbyte of instruction memory.

Phase	Processing time
Full secure boot	ca. 109 seconds
Calculating SHA3 hash	ca. 107 seconds
Verifying signature	1.7 seconds

6.2.2. PMP task encapsulation

To demonstrate that the PMP task encapsulation works, two tasks are created and one of them tries to access a variable that belongs to the other task. The task encapsulation ensures that tasks can only access memory sections that are explicitly assigned to them. This demonstration case is shown in Figure 6.1. The Task1 implements a write operation that writes the value 1000 to the external variable `critical_variable_of_task2`. When compiled, the machine code reveals that this is translated to a store word (`sw`) instruction that tries to write to an address that belongs to the data memory space of Task2. This should not be allowed, as each of the two tasks is only allowed to execute its own code region, and read and write to its own data region.

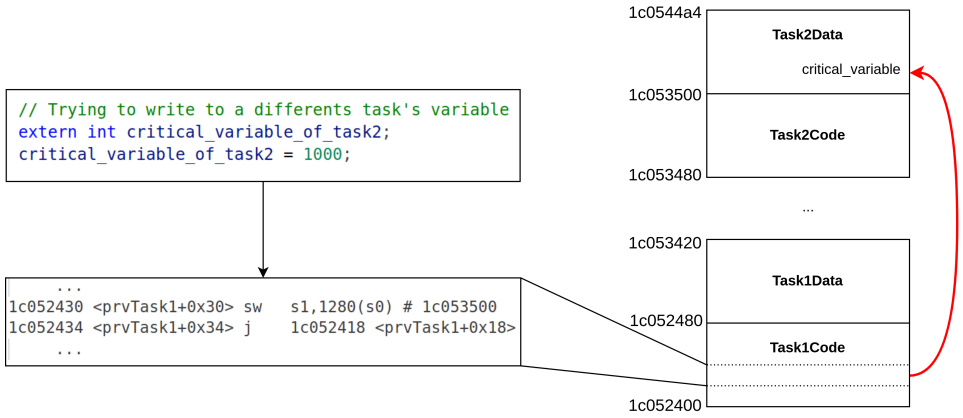


Figure 6.1: Test setup for demonstration of task memory access violation.

The PMP unit of the LSU is responsible for detecting this violation of accessing rights and intervenes in this case. This situation was simulated and Figure 6.2 contains the relevant signals for this situation. The first signal contains the program counter of the core that points to the active instruction. The second signal shows the address that is the input for the PMP unit of the LSU. If the address does not match any of the allowed regions, it sets pmp_err to 1. This does not mean yet that an exception is triggered. An exception is first triggered if the instruction also performs an access request, which is set with the signal exception_in_wb, next signal. Then, the signal for the current privilege level follows,

and the last 7 signals show the address registers of the active PMP regions. At the beginning of the plot, the kernel finishes a system call and enters task 1 at the address 1c004e7e. At this moment, the core switches from M-mode to U-mode because the unprivileged task code is being executed and the PMP unit gets activated. Then the the instruction at address 1c053500 is called that performs the illegal write operation. The graph shows that this instruction causes an interrupt because the PMP unit indicates that this instruction is not allowed and a store operation is requested. The CPU core stops the execution of the task code and enters the trap handler at address 1c010100. With this, the core switches to the M-mode.

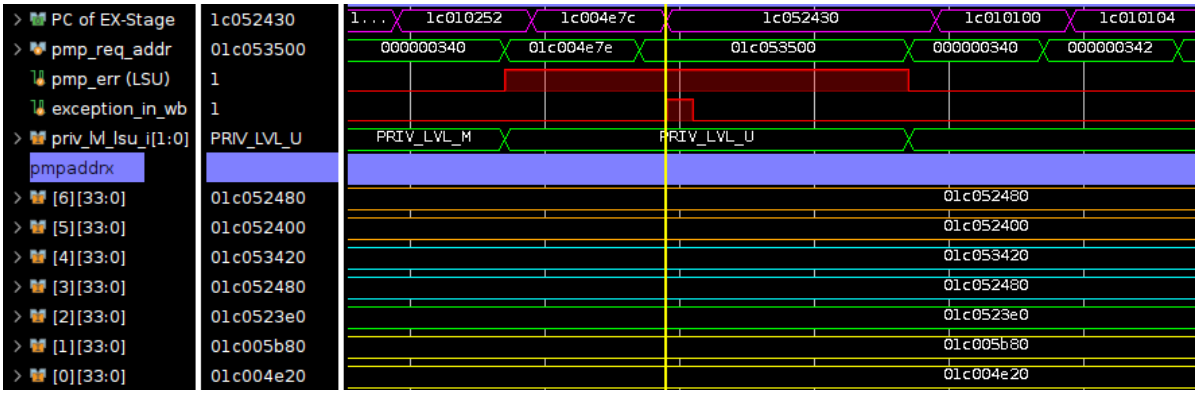


Figure 6.2: Simulation plot during the illegal write instruction.

6.2.3. Dynamic memory allocation with PMP support

The dynamic memory allocation is enhanced by adapting the PMP regions after a memory allocation and deallocation. This section shows how this works in practice. To gain better insights, a minimal program is prepared that allocates 100 bytes of memory on the heap, waits for some time, and calls the free-function to deallocate the memory again. This and the resulting instructions are shown in Figure 6.3. Note, that the wrapper-like system call structure replaces the functions that are part of the kernel by the system call functions. Here, pvPortMallocTask is replaced by MPU_pvPmpMalloc, and vPortFreeTask is replaced by MPU_vPmpFree.

```
void * pAllocatedMemory = pvPortMallocTask( 100 );

vTaskDelayUntil(&xNextWakeTime, pdMS_TO_TICKS(20));

vPortFreeTask( pAllocatedMemory );
```

```
    ...
1c052418 <prvTask1+0x18> li   a0,100
1c05241c <prvTask1+0x1c> jal  ra,1c005b1e <MPU_pvPmpMalloc>
1c052420 <prvTask1+0x20> mv   s2,a0
1c052422 <prvTask1+0x22> li   a1,2
1c052424 <prvTask1+0x24> addi a0,sp,12
1c052426 <prvTask1+0x26> jal  ra,1c004f9e <MPU_vTaskDelayUntil>
1c05242a <prvTask1+0x2a> mv   a0,s2
1c05242c <prvTask1+0x2c> jal  ra,1c005b56 <MPU_vPmpFree>
    ...
```

Figure 6.3: The setup code and the compiled version in assembly language.

Figure 6.4 shows the relevant signals inside the CPU that visualize that the address range for the newly allocated 100 bytes are added to the PMP regions. At the moment when the system call for malloc is called (yellow vertical line), the task requests a service from the kernel, thus, the core switches to M-mode. The kernel is performing the allocation and eventually adds the beginning and end addresses to the PMP regions 7 and 8. After that, the task is entered again and the core switches to U-mode, just to be left again because the task calls a delay function to put itself into the block state.

Later, when the delay is finished, the task is entered again. This can be seen in Figure 6.5, where the task is entered and the function MPU_vPmpFree is called (yellow vertical line). It is visible how the the 6 upper task specific regions are set up before entering the task, including the recently allocated region. Then, right of the yellow vertical line, the previously allocated memory is deallocated, due to the system call, and the two regions 7 and 8 are removed again.

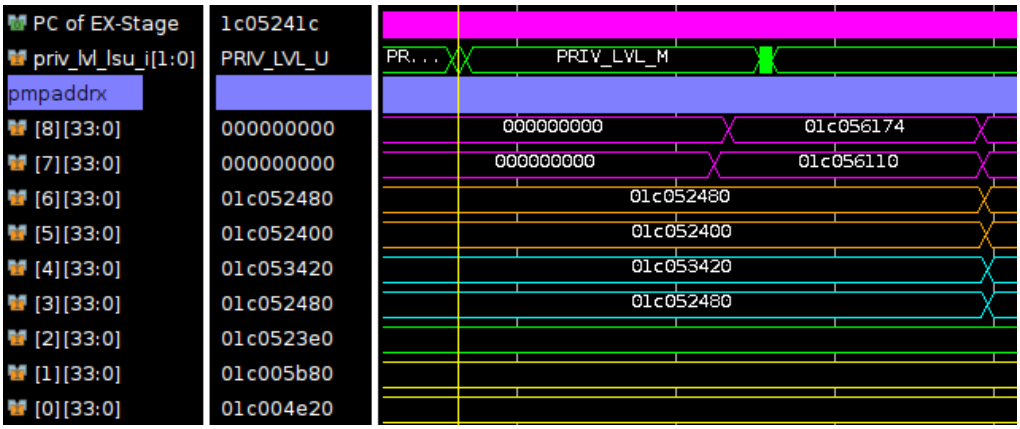


Figure 6.4: Simulation plot capturing the adding the PMP rules in region 7 and 8 for an allocated memory region.

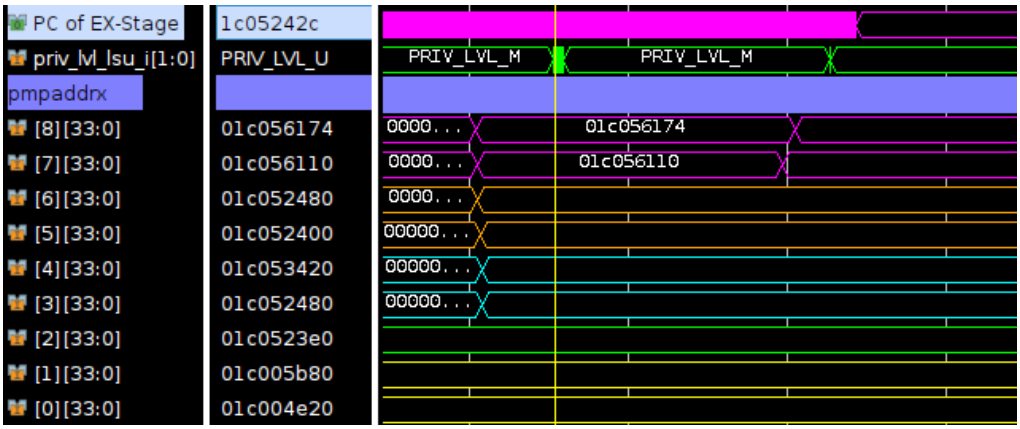


Figure 6.5: Simulation plot capturing removing the PMP rules for a freed memory region.

6.3. Impact of PMP on Real-Time Performance

For RTOSs, it is of utmost importance that tasks finish in time to have certain real-time guarantees. With the increasing complexity of embedded systems, it is necessary to assess the timing overhead induced by the intervention on the RTOS kernel. Otherwise, it is uncertain whether the RTOS is able to meet timing constraints for a given application. For that reason, there are many approaches to benchmarking the real-time performance.

6.3.1. Benchmark Metrics

First of all, embedded applications are often application-specific so, they greatly differ in their requirements. Therefore, having a generic application scenario fails to accurately represent the real performance of a given RTOS. Standardized tests or applications for benchmarking metrics for a particular RTOS is nearly impossible because even with the same hardware setup, different test can lead to different results. This is due to different background operations at varying times, different load conditions, and varying hardware interactions [94]. Instead, this section will evaluate most frequently used RTOS services, also called *fine-grain* benchmarking [95]. This section will evaluate the impact of the PMP-support on the real-time behavior. Only metrics that directly affect the operational performance of FreeRTOS are evaluated. Setup times, such as the task creation times, are not interesting for the

core real-time behavior and will be disregarded. The following three metrics are chosen to obtain a representative estimation over the main features:

- Task Switch Time
- Getting and Releasing a Mutex Time
- Dynamic Memory Allocation and Deallocation Time

The task switch time has been chosen because the task switch is the fundamental operation of the scheduler. The efficiency with which a system can switch between tasks has direct impact on the real-time performance as it adds an overhead of execution time. The added PMP configuration and the system call mechanism adds overhead to task switches. For this reason, it is crucial to evaluate this metric. Mutexes are an essential instrument for managing resource access among multiple tasks. A mutex is assigned to a resource, for example a network interface, and a task has to *take* this mutex before using it. If another task also needs this resource, it tries to take the mutex but it fails because only one task is allowed to hold this mutex at at time. First the first task has to finish its operation and release the mutex. After that other tasks can take the mutex and use the resource. In real-time systems, prolonged waiting times for resource accesses can cause the system to miss deadlines. This thesis implemented dynamic memory support that uses the PMP protection, therefore, it is required to assess the added time overhead for allocating and deallocating memory.

6.3.2. Test Setup

The experiments are run in hardware on the Xilinx PYNQ-Z1 FPGA. For measuring all the times for the metrics above, the internal timer of the SoC is used as a reference counter. In the C-code, a few instructions have been added to write to two pins in the GPIO port. Writing to the GPIOs adds minimal overhead because access takes only one cycle. The functions that write to the pins are defined as "inline", which eliminates the additional cycles typically required for entering and exiting a function. As the access to memory-mapped peripherals by tasks is restricted by default, the memory region for the pins have been added to the task's PMP regions at task creation. An additional logic, from now on called benchmark logic, is added to the design that outputs the counted cycles for the measurements. This is shown in Figure 6.6. This logic counts the cycles based on the two GPIOs, highlighted in orange. These are controlled by the added instructions inside the task's code. The benchmark logic implements a state machine that counts the cycles, and outputs them via the ports "cycles_bm1_o" and "cycles_bm2_o", for the GPIOs respectively. There are two modes of counting the cycles, being controlled by the input "diff_mode_enable". If it is 1, the difference of cycles between the rising and the falling edge of an GPIO pin is measured an reported via the outputs. If it is 0, then the cycle count during the *falling* edge is subtracted from the maximum cycles value, i. e. the value during the timer interrupt. This modes are chosen because they facilitate counting the cycles for the three metrics. The mutex and memory allocation metrics are measured with the first mode, and the context switch time is measured with the second mode.

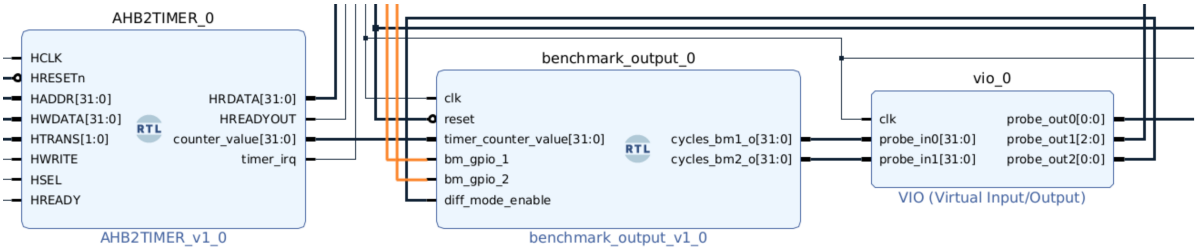


Figure 6.6: Added measurement logic in Vivado.

With Vivado's LogiCORE™IP Virtual Input/Output (VIO v3.0), it is possible to extract the counter values via the JTAG interface to display it in Vivado. As automated data logging is not supported, the values are displayed on the screen in Vivado and the screen is recorded. The task under observation is called every two seconds. For every measurement, 30 samples are recorded from which the mean and standard deviation are calculated. Before recording the cycles, the task is run for 30 seconds

to not start with an empty cache that would significantly increase the cycle count due to the initial cache misses.

Table 6.3: FreeRTOS configuration parameters.

Configuration	Value
configUSE_PREEMPTION	1
configTICK_RATE_HZ	20,000,000
Cache memory access time	1 cycle
BRAM memory access time	50 cycles

6.3.3. Metrics

The following describes the exact method for how the three metrics are measured. The benchmark logic is used to count the cycles based on the two input pins.

Task Switch Time

Task switches can be initiated in two ways. Firstly, the timer interrupt is triggered and the scheduler launches the next task. Secondly, a task sets itself to the blocked state, prompting the scheduler to take over and start the next task. A straightforward method for a task to put itself into the blocked state is the function call `vTaskDelayUntil`, where a parameter specifies after how many SysTick-interrupts the task shall be activated again. Both cases contain the following steps that are essential for a task switch:

- Saving context of interrupted task
- Switch context: Finding next task that is ready
- Restoring context for the next task

In the first case, the timer interrupt triggers a trap that puts the CPU into M-mode and starts the trap handler that performs the steps above. For the second case, `vTaskDelayUntil` is called from within a task. This puts the calling task into the blocked state and issues the system call “task yield” to initiate the steps above. When PMP is used, `vTaskDelayUntil` cannot be called directly because it is a kernel function. Instead, `MPU_vTaskDelayUntil` is used which raises the privileges and then calls the actual `vTaskDelayUntil`. After returning from that function, `MPU_vTaskDelayUntil` resets the privileges again (see section 5.4.4). Moreover, the PMP configuration must be updated during the task switch.

To account for both cases, the task switch times are measured as in Figure 6.7. The purple area is the time to be measured, and contains both task switch situations. For this measurement, an empty task is used that does nothing else than setting a GPIO pin to 1, which is used by the benchmark logic. (Additionally, and LED on the FPGA board is toggled to give a visual feedback. This is not needed for this test and adds a small overhead, but it gives a reassuring visual confirmation that the test is running correctly.) When this task is ready to run, a SysTick-interrupt happens which marks the beginning of the first task switch at time t1. This first task switch ends as soon as the GPIO pin is set to 1. Then, the task immediately sets itself into the blocked state by calling `vTaskDelayUntil`, which marks the beginning of the second task switch at t2. This task switch ends at time t3, at the moment when the kernel launches the next task with the instruction `mret`. The benchmark logic is only able to either report the passed cycles between a rising and falling edge of a GPIO pin, or between the timer reset value (t1) and the falling edge. Therefore, it is not possible to directly measure the time of task switch 1, but only of task switch 2 and the whole time from t1 to t3. The passed cycles for task switch 1 must be indirectly calculated by subtracting task switch 2 from the whole. Task switch 2 is simply calculated by subtracting the Timer Counter (TC) value of t3 from the TC of t2. This seems cumbersome but it keeps the complexity of the measurement setup low. It arises because at the moment of an interrupt, it is not trivial for the benchmark logic to know whether a task switch will follow or not.

This measurement is run for the standard FreeRTOS version that does not add any PMP functionality and no system calls, as well as for the PMP version using 16, 32, and 64 regions. The task switch time is expected to be dependent on the number of created tasks, because the scheduler’s internal list for tasks is longer for more tasks. Therefore, the measurement is run for 1, 2, 4, and 8 tasks. Combined

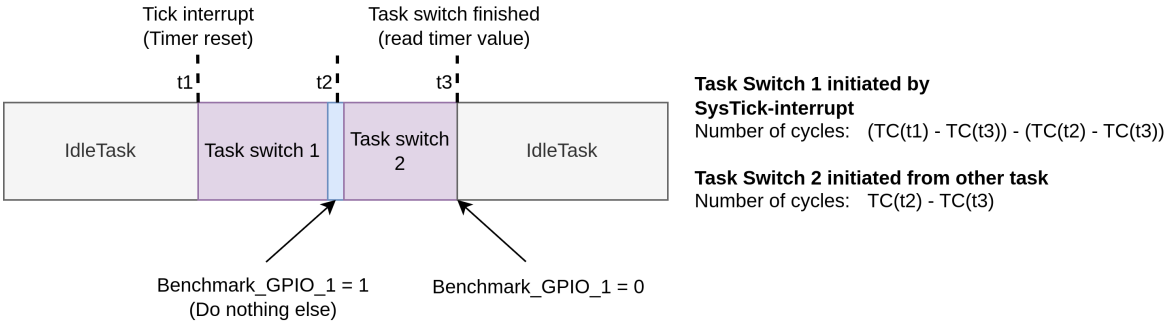


Figure 6.7: Measurement method for task switch time in cycles.

with the varying number of PMP regions, there will be 16 measurements for each of the both task switches. Each measurement consists of 30 recordings.

Taking and Releasing a Mutex Time

The measuring method for taking and releasing a mutex is more straightforward and illustrated in Figure 6.8. Both GPIO pins are used: the first pin for measuring the taking of the mutex, and the second for releasing it. Right before the take and release operation, the GPIO pin is set to 1 and set back to 0 as soon they finish, respectively. During the measurement, only one task is activated that repeatedly takes and releases the mutex. Then the task is delayed for 2 seconds and it repeats. The benchmarking logic reports two times. “cycles_bm1_o” outputs the cycles needed for the taking the mutex and “cycles_bm2_o” outputs the cycles for releasing it.

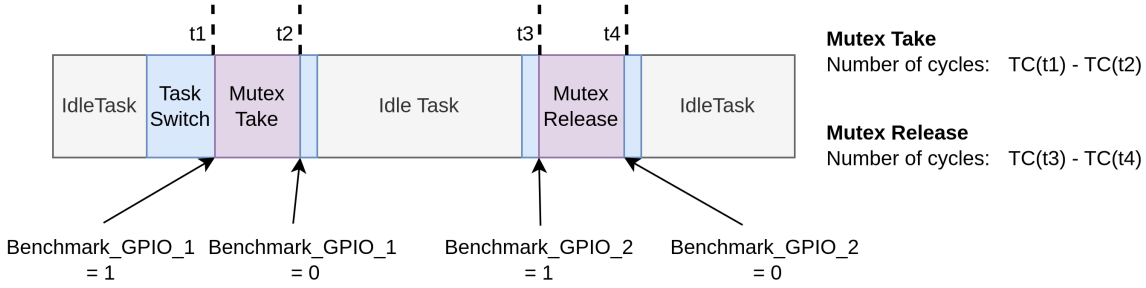


Figure 6.8: Measurement method for taking and giving mutex time in cycles.

This measurement is also run for the standard FreeRTOS version, as well as for the PMP version using 16, 32, and 64 regions. It is not expected that the number of tasks affects the measurements, therefore, these measurements are only run with one 1 task. This results in 4 measurements for each taking and releasing the mutex, again with 30 samples for each.

Dynamic Memory Allocation Time

Very similar, the memory allocation and freeing is measured as in Figure 6.9. During the measurement, again only one task is activated that repeatedly allocates 100 byte of memory on the heap and then deallocates it again. The allocated memory range is saved in the PMP regions 9 and 10 because the first optional regions 7 and 8 are used for the GPIO pins. The benchmarking logic reports two times. “cycles_bm1_o” outputs the cycles needed for the allocation and “cycles_bm2_o” outputs the cycles for freeing these 100 byte again.

Again, the measurements are also run for the standard FreeRTOS version, as well as for the PMP version using 16, 32, and 64 regions. It is not expected that the number of tasks affects the memory allocation time, therefore, these measurements are only run with one 1 task. This results in 4 measurements for each allocating 100 byte of memory on the heap and deallocating it again, again with 30 samples for each.

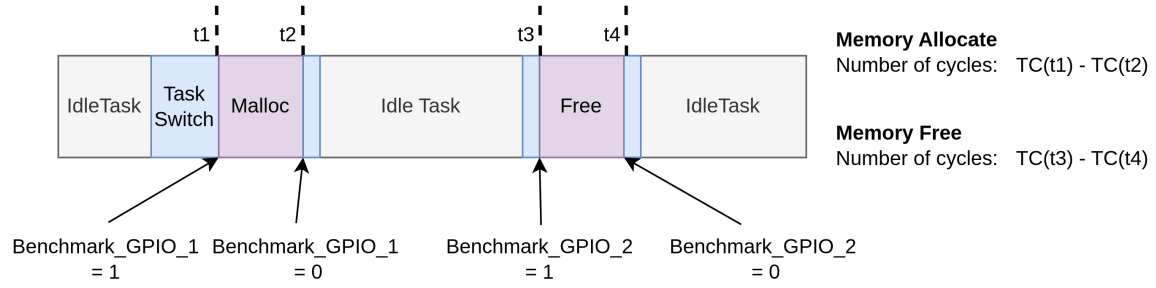


Figure 6.9: Measurement method for allocating and freeing memory time in cycles.

6.3.4. Results

The following figures present the results of the described test cases of the previous chapter. The three test cases defined in the previous chapter were run on the FPGA and 30 cycle count measurements were recorded for every run to obtain a mean and standard deviation for the cycle count.

Task Switch Time

Figure 6.10 and Figure 6.11 show the impact of the number of PMP regions and number of tasks on the task switch time. The first figure for the SysTick initiated task switches contains the mean number of cycles that is the difference of the means of the two measurements explained above. It groups the measurements for the four different number of PMP regions. Unlike the following figures, this one does not include standard deviations because it was not measured directly. In general, it can be seen that the more PMP regions are used, the longer the first task switch takes. For the standard FreeRTOS version without PMP, approximately 6,000 cycles were counted, while for only one task, significantly less with around 4,600 were needed. For 16, 32, and 64 regions the cycle count for one task are 8,700, 10,500, and 13,500, respectively, and with increasing number of tasks, it somewhat rises by up until 600 cycles for each if 8 tasks are active.

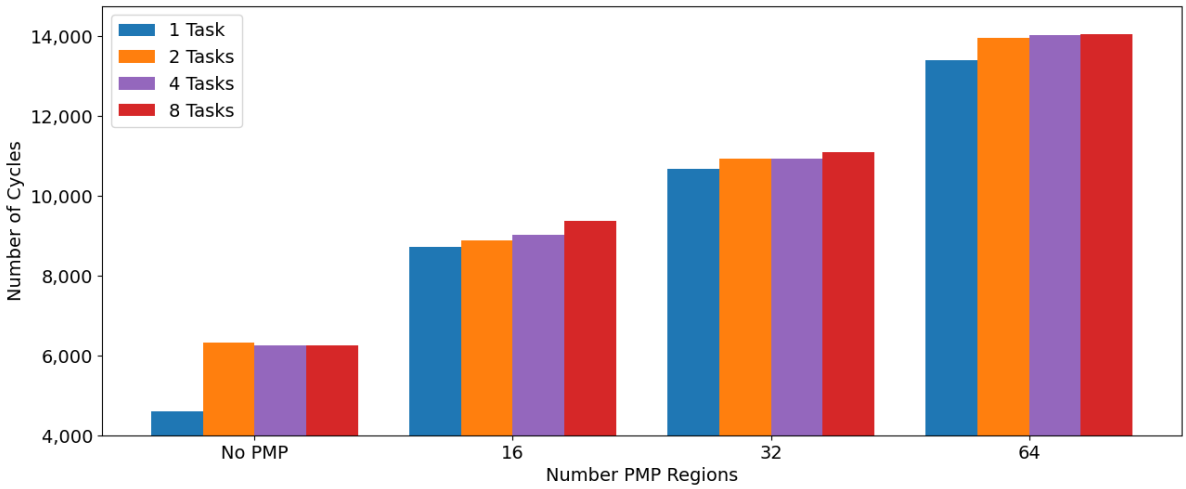


Figure 6.10: Comparison of number of cycles needed for a task switch initiated from SysTick-interrupt for different numbers of PMP regions and tasks.

The second figure contains the mean number of cycles for the measurements for the task-initiated task switch. Additionally, the standard deviation is reported. The counted cycles here are significantly higher than in the previous figure, starting with 8000 cycles for 1 task without using PMP, and gradually rising for more tasks until 9,300 for 8 tasks. Similarly, for 16 PMP regions, the mean cycles start at around 12,700, growing until 13,500. For 32 regions, it starts with 14,600 cycles for one task, but then the value for two tasks with 14,000 is surprisingly lower. However, for 4 and 8 tasks it grows again until 15,300. For 64 regions, the value starts at 18,000 for one task, and rises modestly to 18,600 for 8 tasks. The standard deviation ranges from exactly 0 for 1 task without PMP to 580 for 2 tasks without PMP.

The assumption here is that the variation in the data is mainly caused by the unpredictable hit and miss rates of the caches. This is because the standard deviation tends to be smaller for only one task, where code is fetched from fewer different places in the instruction memory than for more tasks. For the case where the deviation is 0, the same value was measured 30 times, which is probably by chance that it deterministically fits the access patterns of the cache. The main takeaway from standard deviation here is that the measurements lie somewhere close the mean and are therefore reasonably representative.

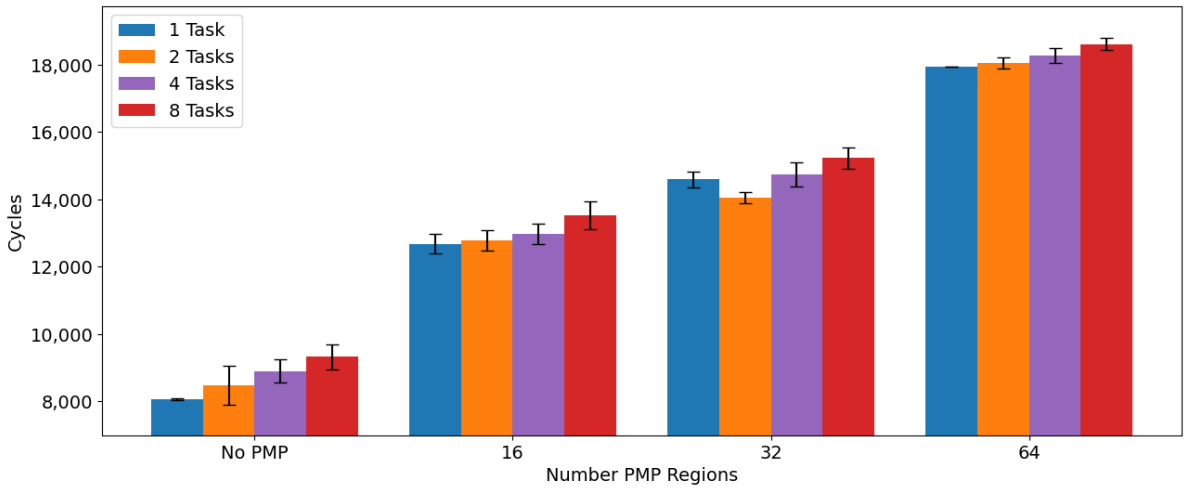


Figure 6.11: Number of cycles needed for a task switch initiated from another task delaying itself. The mean and standard deviation of cycles is reported for different numbers of PMP regions and tasks.

These tasks switches perform many steps and it is not obvious which operation contribute with how many cycles. To gain a better understanding of the composition of the cycles values, Table 6.4 lists a detailed summary of the tasks switches. These values were obtained by simulation in Vivado. For that, the task delay time was reduced to 2 ms, otherwise the simulation would take too long. In simulation, the task switches take a similar number of cycles as when run on the FPGA. For that reason, the simulation results are considered representative. Three cases, each with one active task, were selected: First, with PMP deactivated, second, with 16 PMP regions, and third, with 64 regions. The measurements in simulation were taken after 10 full task switches passed.

The table shows that the total number of cycles for the task switches is roughly similar to the results presented before. The cycles for the SysTick initiated task switch with 4,285 for no PMP, and 9,043 for 16 PMP regions, fit well into the numbers in Figure 6.10. For 64 regions, 14,002 cycles is a little more than expected. For the task-initiated task switch, 7,014 for no PMP is little less than expected when compared to Figure 6.11. 12,352 cycles for 16 regions and 18,184 cycles for 64 regions again fit well. These deviations are assumed to be normal statistical variations as these simulation results contain only one measurement for each case, and not the mean value as in the previous figures. Going from top to bottom through the table, the first store context takes between 671 and 923 cycles. Although increasing for more regions, there is no reason for that. They all run the exact same instructions in this step, therefore, this is assumed to be noise. The same step in the lower half of the table ranges from 907 to 1,120 in a different order, which supports this assumption. The increment tick step takes 1,907 cycles without PMP and approximately 600 cycles more for 16 or 64 regions. The reason for that is that the kernel functions are replaced by the MPU-wrapper functions. At the beginning of these wrapper functions, the kernel checks whether the system is already in privileged mode or in user mode. For the latter, privileges need to be raised. During the increment tick step, the system is already in privileged mode, so raising privileges is not necessary. However, kernel functions are used internally during this step. The added checks at the beginning cause this increase when PMP is used, independent on the number of regions. The switch context step takes between 701 and 1,031 cycles. In the lower half between 781 and 1,118 cycles are needed. This functions does not call other kernel functions internally, therefore, this variation is again considered noise. Then the update of the PMP configuration follows. This step is only used if PMP enabled and it adds a significant overhead depending on the number of regions. Combined with the values in the lower half, this step takes around 2,000 cycles for

16 regions, and around 6,800 for 64 regions. The more regions are used, the more registers need to be configured. This step contributes the most to the overall overhead. After that, the privilege status is updated by writing the following privilege mode the read-only variable `privilege_status`. This step is only used when PMP is used and takes around 400 cycles. Restoring the context is not affected by the PMP feature and takes roughly 1,000 cycles. The SysTick initiated task switch concludes with resetting the privileges before starting the user task. This applies only if PMP is activated and accounts for roughly 1,200 cycles. If the task is initiated by `vTaskDelayUntil` two more steps exist. First, the syscall to raise the privileges which takes around 1,00 cycles and works similar to the previous syscall. And second, the actual function call of `vTaskDelayUntil`. This function suffers the same way as the increment tick step. The function uses kernel function internally and the added privilege mode checks add cycles. Without PMP it takes 3,027 cycles, with PMP it increases by 2,000 to 2,700 cycles. The 700 cycle difference is assumed to be noise. Lastly, there are cycles that cannot be assigned to any of these steps, such as writing to GPIO pins, the function call for entering and leaving the empty task, and toggling an LED as a visual confirmation during the measurements.

Table 6.4: Simulated number of cycles for task switch in detail, comparing no PMP, 16 PMP, and 64 PMP regions.

	Number of cycles		
	No PMP	**16 PMP**	**64 PMP**
Task switch initiated by SysTick	4,285	9,043	14,002
Store context	671	811	923
Increment tick	1,907	2,681	2,406
Switch context	701	975	1,031
Update PMP configuration	-	1,964	6,863
Update privilege status	-	383	454
Restore context	1,006	991	1,085
Syscall: reset privileges	-	1,238	1,240
Task switch initiated by vTaskDelayUntil	7,014	12,352	18,184
Syscall: raise privileges	-	1,056	946
vTaskDelayUntil until task switch	3,927	6,046	6,733
Store context	1,120	907	1,033
Switch context	781	778	1,118
Update PMP configuration	-	2,131	6,848
Update privilege status	-	387	387
Restore context	1,186	1,047	1,119
Empty task (function calls and setting GPIO pin)	*604*	*417*	*479*

Taking and Releasing a Mutex Time

Figure 6.12 presents the cycle time FreeRTOS needs to take and give (release) a mutex of one active task. In every case, the give time is substantially higher than the take time. Without PMP, the taking the mutex takes 1,950 cycles, while giving the mutex takes 4,100 cycles, which is around 2,000 cycles more and double the cycles. For the three other cases, giving takes also takes around 2,000 cycles more, however, each measurement takes around 3,000 more cycles. The number of regions does not seem to have an effect on the cycle times, but the mere fact that PMP is used increases the cycle times by an offset of 3,000 cycles. This is likely the computation overhead for the system calls. The standard deviation without PMP is 70 cycles, increasing to 110 cycles for 16 regions, increasing again to 250 cycles for 32 to regions, and dropping again to 55 for 64 regions. This variation is suspected to be random, however, the 64 regions standard deviation is again curiously lower.

Dynamic Memory Allocation Time

Lastly, Figure 6.13 shows the cycle counts for the last test case, the dynamic memory allocation. Without any PMP support, the allocation and free operations take a similar amount of time with around 6,800 cycles. These are the unmodified implementations of the heap_4.c heap management file that allocate and free 100 byte of memory on this platform. This happens with a small standard deviation of

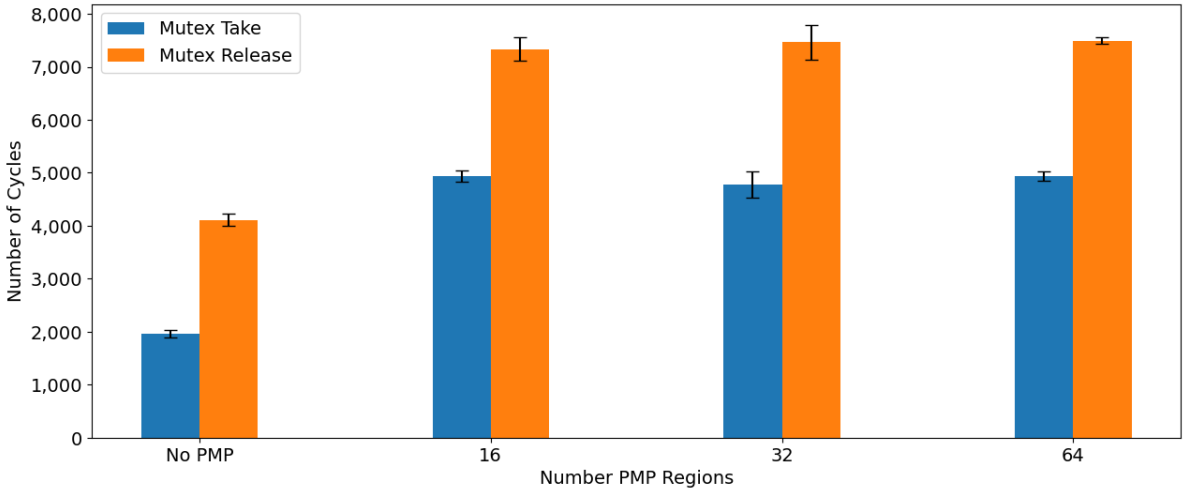


Figure 6.12: Cycles needed for a taking and releasing a mutex, compared for different numbers of PMP regions.

100 and 210 cycles, respectively. When PMP support is added, the cycle times, in general, increase by a factor of 1.72 to around 11,700 cycles for both allocation and deallocation. Hence, the absolute increase in cycles is roughly 3,900. The cycle counts for both operations appear to be independent of the number of PMP regions. For 64 regions, the cycle counts are insignificantly smaller which is likely due to higher cache hit rates. The increase for when PMP is enabled can be explained by two effects. Firstly, the malloc and free operations are called by the task via a system call when PMP is enabled. Similar to the previous test case, the system call adds around 3,000 cycles to the execution. Secondly, adding the PMP regions to the task's TCB when allocating memory, and removing the configuration when deallocating memory, add the residual 900 cycles. In this case, the PMP regions 9 and 10 are used to configure the access rights for the allocated region. It is expected that more cycles are needed when higher regions are used because the added configuration algorithm starts at the lowest regions and stops as soon as it finds an empty region, see subsection 5.4.5.

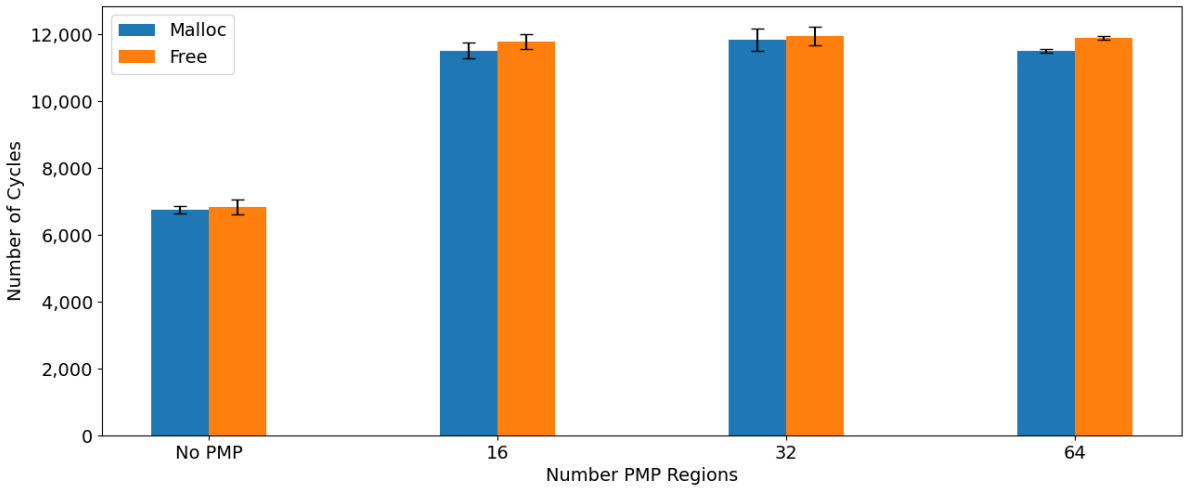


Figure 6.13: Comparison of number of cycles needed for allocating and deallocating memory. The mean and standard deviation is reported for different numbers of PMP regions.

6.4. PMP Encryption Performance

This section briefly presents the performance of the encryption unit when used for encrypting the code and/or data of a task. The detailed performance measurement for the integrated encryption unit are extensively performed and described in [21] already. When integrated into this project, a comprehen-

sive evaluation of the encryption unit's performance requires conducting a wide range of detailed test cases across various applications. However, such an extensive assessment is beyond the scope of this thesis. For this thesis, only a small comparison of a task repeatably multiplying 1000 values with and without encryption is evaluated here. This offers a rough orientation on the encryption unit's impact on computational performance under these specific conditions. Listing 6.1 contains the code used for the measurement. The code calculates the square values of all the number from 0 to 1000 and writes each of them to an array that hold a thousand entries. To increase the likelihood of cache misses, that result in a encryption or decryption operation, the original value is written at the position of the square result modulo 1,000.

Listing 6.1: Simple test code for encryption time evaluation.

```
/* ... */
for (;;)
{
    set_benchmark_gpio_1();
    volatile uint32_t array_1[1000];

    for (uint32_t k = 0; k<1000; k++)
    {
        uint32_t result = k*k;
        array_1[result % 1000] = k;
    }
    reset_benchmark_gpio_1();

    vTaskDelayUntil(&xNextWakeTime, pdMS_TO_TICKS(2000));
}
```

The recorded cycles times are presented in Figure 6.14. Without any encryption, the code from above takes around 171,500 cycles with a standard deviation of 850. With the encryption enabled for the whole task, the calculation takes 3.4% longer by taking 177,000 cycles. The implementation integrates two separate encryption units for code memory and data memory. Hence, it is possible to only encrypt the runtime data. The cycles needed for only encrypting the data memory is marginally less with 177,100 cycles. The standard deviation without encryption is 850 cycles, and for the both other cases, it is around 1,200 cycles. However, the result for only data encryption is not very meaningful because the task's program code is very small. Probably, not many accesses to the external instruction happen anyway. Further research is needed to investigate the performance overhead comparison between only data and full encryption.

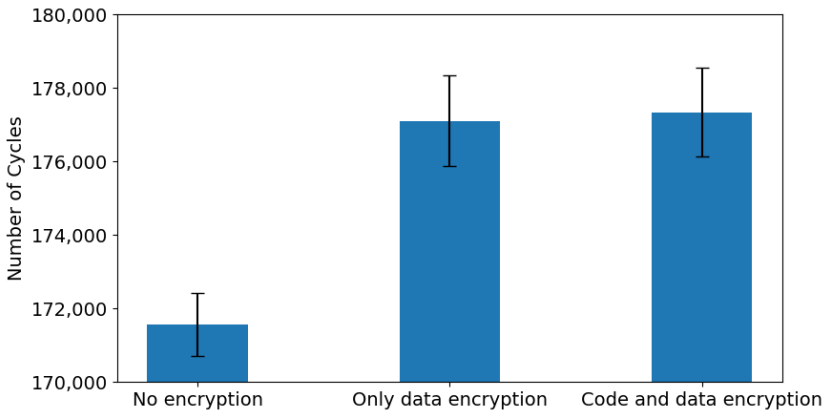


Figure 6.14: Comparison of cycle counts for the execution of an example test application for using the encryption unit.

6.5. Area overhead

The PMP feature of RISC-V is implemented in hardware, which inevitably leads to more area usage of the processor. On top of that, enabling PMP makes it mandatory to implement the U-Mode privilege level, further increasing the area. The CV32E40S RISC-V core was synthesized and implemented in Vivado with different PMP settings. The Table 6.5 gives an overview of how many Look Up Tables (LUTs) and registers were needed for the implementation of each configuration and also identifies the internal units responsible for this usage. The numbers were obtained from the utilization reports that Vivado provides after the implementation of the design.

Table 6.5: Comparison of used area for FPGA implementation when implementing different numbers of PMP regions.

	Slice LUTs	Slice Registers	DSPs
RISC-V core without PMP	4,391	2,294	3
IF-stage	972	358	0
ID-stage	1,524	242	0
EX-stage	730	283	3
WB-stage	4	4	0
Control and Status Registers	125	328	0
Load-store-unit	137	44	0
Others	899	1,035	0
Core with 16 PMP regions	9478	2949	3
IF-stage	1,560	352	0
ID-stage	1,885	241	0
EX-stage	794	282	3
WB-stage	3	4	0
Control and Status Registers	3,980	971	0
Load-store-unit	442	43	0
Others	814	1,056	0
Core with 32 PMP regions	14,718	3,564	3
IF-stage	2,259	357	0
ID-stage	2,089	242	0
EX-stage	854	282	3
WB-stage	4	4	0
Control and Status Registers	7,958	1,616	0
Load-store-unit	800	43	0
Others	754	1,020	0
Core with 64 PMP regions	23,592	4,856	3
IF-stage	2,055	352	0
ID-stage	3,314	243	0
EX-stage	1,035	282	3
WB-stage	3	4	0
Control and Status Registers	16,245	2,896	0
Load-store-unit	159	43	0
Others	781	1,036	0

When comparing the area of the total core, it becomes apparent that the use of PMP increases the area dramatically. Without PMP, 4,391 LUTs and 2,294 registers are used for the whole core. However, with 16 PMP regions, 9,478 LUTs and 2,949 registers are needed. The number of LUTs is more than doubled for 16 regions, and the number of registers grew by almost 30%. This trend continues when more regions are enabled: For 64 regions, 23,592 LUTs and 4,856 registers are used. The area of this design is more than 5 times bigger in terms of LUTs, and more than double as big in terms of registers, compared to the core without PMP. The numbers also show that most of the area can be attributed to

the control and status register unit. This is the unit, in which the PMP configurations and addresses are stored. The number of LUTs for this unit appears to grow proportionally with the number of PMP regions. The registers also grow with the increasing number of PMP regions, however, somewhat slower. The number of PMP regions also affects the IF-stage and the Load-Store-Unit (LSU), the both components that eventually apply the PMP rules. Without PMP, the IF-stage uses 972 LUTs , growing fast to 1,560 for 16 regions, and 2,259 LUTs for 32 regions, while the registers stay constant at around 355 registers. Curiously, the LUTs go down to 2,055 LUTs for 64 regions. Something very similar can be observed for the LSU. Starting at 137 LUTs without PMP, it goes up significantly to 442 for 16 regions, continues to rise to 800 LUTs for 32 regions, to then suddenly plummet to 159 LUTs when 64 regions are used. No obvious reason has been found to explain this behavior. Vivado's utilization report further reveals that the main sub-unit in the IF-stage and LSU is a unit called pmp_i. This unit is connected to the control and status register unit and holds all PMP address and configurations. However, for 64 regions, this unit is not listed as a subcomponent anymore and is there not part of neither the IF-stage the the LSU. A possible explanation could be that this sub-unit is optimized out into different units in the design but this could not be verified.

However, the SoC also consists of other components and not only the core. The utilization report of the total design for 16 regions is summarized in Table 6.6. The report reveals that the RISC-V core is responsible for 47.4% of the design in terms of LUTs, and 16.0% in terms of registers. The both caches with 6,842 LUTs and 10,896 registers account for 34,2% of the LUTs and 76.1% of the registers. The encryption units and other components, for example peripherals, account for the rest. It is important to consider that the caches were deliberately chosen to be very small for faster synthesis times. In real-world applications the caches would be much larger, reducing the relative area of the core.

Table 6.6: Overview of used area of the SoC design for 16 PMP regions.

Component	Slice LUTs	Slice Registers	DSPs
RISC-V core	9,478	2,294	3
2x Cache	6,842	10,896	0
2x Encryption unit	2,852	1,092	0
Others	807	34	0

6.6. Discussion

This thesis successfully implements PMP support for FreeRTOS, which has been demonstrated in this chapter. The implementation is able to restrict tasks to their dedicated memory regions, and memory access violations to unauthorized memory addresses are blocked by the system. This also includes unauthorized accesses to memory-mapped peripherals. The additional features, the secure boot and the integrated encryption unit also work as intended. Based on the experiments, the following findings and remarks can be emphasized:

- **Performance overhead** The results show that the integration of PMP comes with significant performance overheads that must be considered for the design of real-time applications. However, the use of PMP can still be reasonable efficient because it does not shift the performance to a different order of magnitude. In the extremest case for the task switch performance, using all 64 regions for 1 task, increases the task switch time by a factor of 2.5. Considering that the task switch is a relatively quick operation compared with the actual task code, and considering the huge security benefit on the system, integrating the PMP support for FreeRTOS can be definitely be a viable option for real-time applications.
- **System calls overhead** The system call mechanism, separating kernel space from user space, is a necessary consequence of using PMP. This is because user tasks needs to request services that should only by accessible to the kernel. This has an overarching effect on the real-time performance because all kernel functions are slowed down. The measured results for the mutex handling are an example for this. The system calls for mutex give and take cause both operations to take double the amount of cycles. It is expected that the absolute system call overhead will be constant for all kernel services, therefore, smaller functions will suffer more from this overhead

than functions that need more cycles to finish.

- **Area overhead** While the overhead on execution performance is rather modest, the area overhead of using PMP is immense. Even when using 16 regions the area of the total core doubled, when considering LUTs. With higher numbers of regions, the effect is even more pronounced, as the number of LUTs used grows up to a factor of 5, and the number of registers doubles, when using 64 PMP regions. Whether or not this could be a deal breaker for a real-world application depends on the design of the whole SoC. If the design incorporates caches, the increase in core size might not have a significant impact on the whole area. However, very small embedded systems that are designed for having a minimal area and being low in power consumption might not use caches. This rapid increase in area for an additional security feature could unacceptable for these applications.
- **Secure boot start-up time** Although the secure boot is not the main focus of this thesis, implementing a secure boot in software as done in this thesis might not be a good option for devices that require a small start-up time. The full secure boot takes 109 seconds for a memory range of 1.35 Mbyte at a clock frequency of 20 MHz. Given a real-world application that runs at 100 MHz, it would still spend 22 seconds on the secure boot of a relatively small application of 1.35 Mbyte that mainly contains the FreeRTOS kernel. If this is not acceptable, hardware acceleration support can be considered.
- **Combining encryption and PMP** Augmenting the PMP mechanism with an encryption unit is an elegant way to enable selective task encryption. The already present PMP address matching functionality is leveraged to temporarily enable or disable the encryption unit. The recorded cycle times show that enabling full encryption for both code and data memory leads to a 3.4% increase in execution time, rising from 171,500 cycles to 177,000 cycles. This is a relatively small increase which indicates that integrating an encryption unit in this design could be an efficient security improvement. However, as stated before, the data here is too tenuous to derive general conclusions. Therefore, further exploration is needed here.

6.6.1. Limitations

Further, while these results provide valuable insights, it is also important to acknowledge certain limitations of the interpretation of these results. Additionally, when considering the broader context, this work has limitations, which are highlighted in the following section.

Limited Scope of PMP Support in FreeRTOS Implementation FreeRTOS is a comprehensive real-time operating system that offers more functionality beyond the scope of this thesis. For this thesis mainly dealt with the most fundamental components of FreeRTOS, such as the scheduler, the resource and memory management, or the trap handling. However, for real world applications more advanced features are needed, like inter-task communication, more advanced interrupt handling, and features like stream buffers or event groups, all of which are also supported by FreeRTOS. While this thesis has established the groundwork, there remains significant work to be done to achieve a robust and mature implementation.

System call handling The system call handling concept is implemented as in the official MPU-FreeRTOS V9.0.0 release for ARM platforms. FreeRTOS later released redesigned system call handling mechanism because the previous one suffered from severe security flaws. This thesis did not adapt the redesigned version because it is significantly more complex and transferring it to RISC-V proved to be challenging. Instead, the previous version was implemented and improved as described in subsection 5.4.4. Although offering protection against simple control flow attacks, it is still possible to compromise the kernel for a skilled attacker that knows about the memory structure.

More exploration of encryption unit performance needed The performance measurements of the memory encryption unit was evaluated with only a simple example which limits the results in its interpretability due to the insufficient amount of data available.

7

Conclusion

This final chapter summarizes the work done in this thesis and discusses possibilities for future work. A brief summary of each chapter is given in section 7.1, and future work is discussed in section 7.2

7.1. Summary

The goal of this thesis was to develop a platform that implements PMP support for a RISC-V core running FreeRTOS. Bringing together one of the most popular real-time operating systems, FreeRTOS, and security for RISC-V platforms has been neglected by the industry and open-source community. This is concerning, because embedded devices get increasingly more connected and often run untrusted third-party software. This has led to damage on multiple occasions in the past. The thesis examined two weaknesses of resource-constraint devices: First, the flat memory structure inherent to FreeRTOS is prone to being readily exploited and compromised by attackers. And second, data and program code on an external memory unit can be read out and modified because the pins are accessible. Based on these weaknesses, a secure platform was proposed and designed that demonstrates a holistic approach to integrate FreeRTOS with PMP support into a System-on-Chip. This thesis made various contributions: First, a secure System on Chip (SoC) was developed that integrates the CV32E40S core from OpenHW Group. This is a state-of-the-art implementation of the latest RISC-V specification. Second, a FreeRTOS solution was developed that leverages RISC-V's Physical Memory Protection (PMP) feature to secure malicious tasks. Furthermore, a software based secure boot procedure for this system was developed. Then, an existing lightweight encryption unit was integrated and combined with the PMP feature. This encryption unit implements the Prince encryption scheme to encrypt the data stored on the external memory unit. Unused bits in the PMP register were used to enable the encryption unit for external memory encryption. This made selective memory encryption for chosen PMP regions possible. Additionally, the encryption unit's security was improved by adding address-dependent encryption to avoid determinism between memory cells. Apart from this, this thesis offered an general overview of known attacks on embedded devices. Lastly, the proposed design was implemented on a Xilinx PYNQ-Z1 FPGA platform to investigate the effect the PMP support for FreeRTOS has on the real-time performance. Moreover, the effectiveness of the implemented security features was validated, and the area overhead was reported. Below follows a summary of the content for each chapter:

Chapter one presented an overview of the thesis subject, highlighting its importance and the related research, followed by outlining the main contributions made by the thesis.

Chapter two provided the reader with the necessary background information needed for understanding this thesis. It began with an introduction to embedded system architectures in general, and continued by explaining the instruction set architecture RISC-V. Then, it gave relevant background information on the memory architecture, followed by explaining FreeRTOS in detail. Finally, it explained the C-toolchain, which was needed to understand the implementation in later chapters more easily.

Chapter three provided a comprehensive background overview of embedded system security. It began with an explanation of the core security aspects: integrity, confidentiality, and availability. Following this, it discussed physical attacks that target the hardware components of embedded systems. The chapter then delved into software attacks, which exploit vulnerabilities in the system's software, followed by providing details about network attacks. The chapter then explained four attack scenarios in, which served as the primary focus of this thesis, and concluded with an overview of current countermeasures.

Chapter four proposed a high-level secure system design that was the basis for the rest of this thesis. For that, a representative system architecture running FreeRTOS was assumed and analysed for possible attacks. Based on these attacks, requirements were derived that would make the system secure if fulfilled. The requirements were derived in two steps: First, high-level cybersecurity goals were defined that specify a need while being solution agnostic. In the second step, specific feature requirements were developed that formed the foundation for the design in the following chapter.

Chapter five presented the design choices and implementation of the secure system presented in the previous chapter. It began with the evaluation of different available RISC-V cores and selected the one the fitted the needs of this design best. This was followed by the memory layout and hierarchy of the design. After that, the secure boot to ensure the integrity of the FreeRTOS kernel was presented, followed by the design and implementation of using RISC-V's PMP feature FreeRTOS to restrict tasks. Lastly, the integration combination of the memory encryption unit with the PMP feature to selectively encrypt FreeRTOS tasks was presented.

Chapter six demonstrated the added measures are implemented correctly and evaluated the performance of selected features. First, the experimental setup was explained to the reader, and then, test cases were described and conducted which demonstrated the correct workings of the implemented measures. After that, the impact on the real-time capability of the PMP integration into FreeRTOS was inspected in detail with different example test setups. Then, the performance of the encryption unit was evaluated when used for FreeRTOS tasks. As PMP is a hardware feature of RISC-V, the additional area of its use was reported. Lastly, the results and findings were discussed.

7.2. Future Work

This thesis developed a working FreeRTOS implementation with PMP support that implements the most essential features of FreeRTOS. FreeRTOS is a versatile operating system that, despite aiming to be minimal, still implements many features and modes that this thesis has not explored. Examples being advanced inter-task communication, more sophisticated interrupt handling, or a tick-less mode for low power applications. More work is needed to explore these additional features, integrate them with the PMP support. Beyond these immediate tasks, this thesis has also identified further areas that require additional research:

- **Improving the system call architecture for higher security** The system call handling is a core mechanism to ensure a secure separation of kernel and user space. This thesis adopted the system call concept of the official MPU-FreeRTOS V9.0.0 release for ARM platforms. This mechanism has inherent security flaws that this thesis reduced but could not fully eliminate. Future work has to investigate improved system call handling mechanisms. Either it adopts the improved official implementation for MPU-FreeRTOS for ARM platforms or a new solution needs to be developed.
- **Improving the area usage of core for PMP regions** Enabling PMP for the CV32E40S RISC-V core significantly increased the core's area. In terms of LUTs, using 16 PMP regions roughly doubles the core's area, while 64 PMP regions increase it by a factor of 5. When used in an SoC with caches and peripherals, as in this thesis, this increase might not be dramatic because it remains relatively small, particularly in comparison to the size of the caches. However, for smaller designs that do not use caches, this increase might be unacceptable. Future work could focus on optimizing the area usage for PMP regions to make the core more suitable for a wider range

of applications.

- **Adding support for different addressing modes for ranges** This thesis mainly used the Top-of-Range (TOR) addressing mode to protect memory regions with PMP. This mode needs two PMP regions to specify a given address range: one PMP region defines the upper end of the memory range and the previous PMP region defines the start of range. The Naturally-Aligned-Power-of-Two (NAPOT) addressing mode can protect a memory region with only one PMP region. More advanced memory layout and encapsulation schemes can be developed that can use PMP regions more efficiently by leveraging this mode.

References

[1] Precedence Research. *Internet of Things (IoT) Market Size to Hit $3,152.17 Bn by 2033*. https://www.precedenceresearch.com/internet-of-things-market. Accessed: 2024-08-04. 2023.

[2] C Ene. *10.5 Trillion Reasons Why We Need A United Response To Cyber Risk*. https://www.forbes.com/sites/forbestechcouncil/2023/02/22/105-trillion-reasons-why-we-need-a-united-response-to-cyber-risk/. Accessed: 2024-07-13. 2023.

[3] CNBC. *Microsoft-CrowdStrike issue causes 'largest IT outage in history'*. https://www.cnbc.com/2024/07/19/latest-live-updates-on-a-major-it-outage-spreading-worldwide.html. Accessed: 2024-07-31. 2024.

[4] Ionut Ilascu. *Fitbit Gallery Can Be Used to Distribute Malicious Apps*. BleepingComputer. Accessed: 23-07-2024. 2020. URL: https://www.bleepingcomputer.com/news/security/fitbit-gallery-can-be-used-to-distribute-malicious-apps/.

[5] Daily Mail Reporter. "Is your car safe? Jeep hackers that prompted recall of 1.4 million vehicles reveal new hack to control a car at high speeds". In: *Daily Mail* (2016). Accessed: 23-07-2024. URL: https://www.dailymail.co.uk/sciencetech/article-3724472/Is-car-safe-Jeep-hackers-prompted-recall-1-4-million-vehicles-new-hack-control-high-speeds.html.

[6] Zohaib Ahmed et al. "Protecting IoTs from Mirai Botnet Attacks Using Blockchains". In: *2019 IEEE 24th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*. 2019, pp. 1–6. DOI: 10.1109/CAMAD.2019.8858484.

[7] AspenCore Media embedded.com. *The Current State of Embedded Development*. https://www.embedded.com/wp-content/uploads/2023/05/Embedded-Market-Study-For-Webinar-Recording-April-2023.pdf. Accessed: 2024-04-16. May 2023.

[8] The Linux Foundation. *Real Time Linux*. https://wiki.linuxfoundation.org/realtime/start. Accessed: 2024-07-08.

[9] Adam. "A Simple Guide to ARM vs. RISC-V vs. x86". In: *PiCockpit* (Oct. 2023). URL: https://picockpit.com/raspberry-pi/arm-vs-risc-v-vs-x86/.

[10] David Kanter. "RISC-V offers simple, modular ISA". In: *Microprocessor Report* 1 (2016), pp. 1–5.

[11] European High Performance Computing Joint Undertaking. *The European Processor Initiative (EPI)*. Accessed: 2024-07-11. 2024. URL: https://eurohpc-ju.europa.eu/research-innovation/our-projects/european-processor-initiative-epi_en.

[12] Semico Research. "Semico Forecasts Strong Growth for RISC-V". In: *Design & Reuse* (Nov. 2019). URL: https://www.design-reuse.com/news/47182/risc-v-forecast.html.

[13] Gary Mullen and Liam Meany. "Assessment of Buffer Overflow Based Attacks On an IoT Operating System". In: *2019 Global IoT Summit (GIoTS)*. 2019, pp. 1–6. DOI: 10.1109/GIOTS.2019.8766434.

[14] FreeRTOS. *Using FreeRTOS-MPU to Create a Memory Protected System*. Accessed: 2023-07-28. 2023. URL: https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html.

[15] FreeRTOS. *FreeRTOS Kernel History*. https://www.freertos.org/History.txt. Accessed: 2024-07-04.

[16] FreeRTOS. *FreeRTOS-MPU: ARM Cortex-M3 and ARM Cortex-M4 Memory Protection Unit support in FreeRTOS*. https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html. Accessed: 2024-07-04.

[17] SiFive. *FreeRTOS-metal*. https://github.com/sifive/FreeRTOS-metal/tree/9e8a7a3e8e9ef1e256f93c5681a5578ac53eeed4. Commit 9e8a7a3e8e9ef1e256f93c5681a5578ac53eeed4. Accessed: 2024-04-07. 2024.

[18] SiFive. *Example FreeRTOS PMP Blinky*. https://github.com/sifive/example-freertos-pmp-blinky. Accessed: 2024-07-28. 2023.

[19] Zephyr Project Contributors. *Zephyr Project: Releases*. https://github.com/zephyrproject-rtos/zephyr/releases. Accessed: 2024-07-05. 2024.

[20] Dayeol Lee et al. "Keystone: An open framework for architecting trusted execution environments". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–16.

[21] Hans Okkerman. "Embedded Memory Security". Master's Thesis. Delft University of Technology, Nov. 2020. URL: https://repository.tudelft.nl/islandora/object/uuid:4fef19f2-51c8-4210-9e9e-4f5e531de432?collection=education.

[22] Market.us. *Embedded Systems Market Size, Trends | CAGR of 6.8%*. https://market.us/report/embedded-systems-market/. Commit 9e8a7a3e8e9ef1e256f93c5681a5578ac53eeed4. Accessed: 2024-04-07. 2023.

[23] Ji Su Park and Jong Hyuk Park. "Future Trends of IoT, 5G Mobile Networks, and AI: Challenges, Opportunities, and Solutions". In: *J. Inf. Process. Syst.* 16.4 (2020), pp. 743–749. DOI: 10.3745/JIPS.03.0146. URL: https://doi.org/10.3745/JIPS.03.0146.

[24] Daniele Lacamera. *Embedded Systems Architecture: Discover Software Programming on Embedded Devices to Build Safe and Secure Connected Systems*. Birmingham: PACKT Publishing Limited, 2023. ISBN: 978-1-80323-954-5.

[25] Yale N Patt and Sanjay J Patel. "Introduction to Computing Systems; Bits & Gates to C/C++ and Beyond". In: (2020).

[26] Shashank Priya and Daniel J Inman. *Energy harvesting technologies*. Vol. 21. Springer, 2009. ISBN: 978-0-387-76463-4.

[27] ARM Ltd. *AMBA AHB-Lite Protocol Specification*. Accessed: 2024-07-26. 2021. URL: https://developer.arm.com/documentation/ihi0033/c.

[28] Joseph Yiu. *The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors*. Newnes, 2014. ISBN: 978-0-12-408082-9.

[29] Richard Barry. *Get to know FreeRTOS from the Creator!* Accessed: 28-07-2024. 2013. URL: https://www.youtube.com/watch?v=1oagM_tEyeA.

[30] Richard Barry and The FreeRTOS Team. *Mastering the FreeRTOS™ Real Time Kernel: A Hands-On Tutorial Guide*. Release Version 10.6.2. Available at: https://www.freertos.org/Documentation/02-Kernel/07-Books-and-manual/01-RTOS_book (Accessed: 9th March 2024).

[31] Alan Holt and Chi-Yu Huang. *Embedded Operating Systems: A Practical Approach*. Second edition. Cham, Switzerland: Springer, 2018. DOI: 10.1007/978-3-319-72977-0. URL: https://doi.org/10.1007/978-3-319-72977-0.

[32] Michael Opdenacker. *Embedded Linux from Scratch in 50 Minutes (on RISC-V)*. https://bootlin.com/pub/conferences/2024/risc-v/embedded-linux-from-scratch-riscv.pdf. Bangalore: Bootlin, 2024.

[33] Zhang Xiaodong and Zhang Jie. "Design and implementation of smart home control system based on STM32". In: *2018 Chinese Control And Decision Conference (CCDC)*. 2018, pp. 3023–3027. DOI: 10.1109/CCDC.2018.8407643.

[34] RISC-V International. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. Version: 20191213. RISC-V International, 2019. URL: https://riscv.org/technical/specifications/.

[35] RISC-V International. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. Version 20211203. RISC-V International, 2021. URL: https://riscv.org/technical/specifications/.

[36] Rich Wawrzyniak. *RISC-V Market Analysis Report; Application Forecasts in a Heterogeneous World 2024*. Tech. rep. Available at: https://theshdgroup.com (Accessed: 9th March 2024). The SHD Group, Jan. 2024.

[37] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012. ISBN: 978-0-12-383872-8.

[38] David A. Patterson and John L. Hennessy. *Computer Organization and Design - The Hardware/Software Interface (4th Edition)*. Elsevier, 2009. Chap. 5.4 Virtual Memory. ISBN: 978-0-12-374493-7.

[39] MITRE. *CVE-2014-0160*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160. Accessed: 2024-06-18. 2014.

[40] Hyeonguk Jang et al. "MMNoC: Embedding memory management units into network-on-chip for lightweight embedded systems". In: *IEEE Access* 7 (2019), pp. 80011–80019.

[41] FreeRTOS. *Using FreeRTOS on RISC-V*. Accessed: 2024-02-11. 2024.

[42] Giorgio Buttazzo. *Hard Real-Time Computing Systems*. Springer Nature Switzerland, Jan. 2024. DOI: 10.1007/978-3-031-45410-3.

[43] Richard Barry. *Developing with FreeRTOS and RISC V*. https://www.youtube.com/watch?v=ZzPXEOK_sGI. Accessed: 2024-07-30. 2021.

[44] GNU. *CPP manual*. Accessed: 2024-07-11. URL: https://gcc.gnu.org/onlinedocs/cpp/index.html#SEC_Contents.

[45] GNU Project. *GCC, the GNU Compiler Collection*. Accessed: 2024-07-11. 2024. URL: https://gcc.gnu.org/.

[46] Steve Chamberlain. *The GNU linker*. Accessed: 2024-07-11. 1994. URL: https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_mono/ld.html.

[47] George Loukas. *Cyber-physical attacks: A growing invisible threat*. Butterworth-Heinemann, 2015. ISBN: 978-0128012901.

[48] Mark Randolph and William Diehl. "Power side-channel attack analysis: A review of 20 years of study for the layman". In: *Cryptography* 4.2 (2020), p. 15.

[49] Mario Noseda and Simon Künzli. "Check for updates Attacking Secure-Element-Hardened MCUboot Using a Low-Cost Fault Injection Toolkit". In: *Innovative Security Solutions for Information Technology and Communications: 16th International Conference, SecITC 2023, Bucharest, Romania, November 23–24, 2023, Revised Selected Papers*. Vol. 14534. Springer Nature. 2024, p. 126.

[50] Aakash Gangolli, Qusay H Mahmoud, and Akramul Azim. "A systematic review of fault injection attacks on IOT systems". In: *Electronics* 11.13 (2022), p. 2023.

[51] Thomas Korak et al. "Clock Glitch Attacks in the Presence of Heating". In: *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2014, pp. 104–114. DOI: 10.1109/FDTC.2014.20.

[52] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. "Fault attacks on secure embedded software: Threats, design, and evaluation". In: *Journal of Hardware and Systems Security* 2 (2018), pp. 111–130.

[53] Job De Haas. "20 ways past secure boot". In: *Hack in the Box Security Conference*. 2013.

[54] Huanyu Wang et al. "Probing Attacks on Integrated Circuits: Challenges and Research Opportunities". In: *IEEE Design & Test* 34.5 (2017), pp. 63–71. DOI: 10.1109/MDAT.2017.2729398.

[55] Mohammad Tehranipoor and Cliff Wang. *Introduction to hardware security and trust*. Springer Science & Business Media, 2011. ISBN: 978-1-4419-8079-3.

[56] Quoc-Dung Ngo et al. "A survey of IoT malware and detection methods based on static features". In: *ICT express* 6.4 (2020), pp. 280–286.

[57] Austen Knapp et al. "Should Smart Homes Be Afraid of Evil Maids?: Identifying Vulnerabilities in IoT Device Firmware". In: *2024 IEEE 14th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE. 2024, pp. 0467–0473.

[58] Dorottya Papp, Zhendong Ma, and Levente Buttyan. "Embedded systems security: Threats, vulnerabilities, and attack taxonomy". In: *2015 13th Annual Conference on Privacy, Security and Trust (PST)*. ieee. 2015, pp. 145–152.

[59] Yoongu Kim et al. “Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors”. In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 361–372.

[60] Nate Lawson. “Side-channel attacks on cryptographic software”. In: *IEEE Security & Privacy* 7.6 (2009), pp. 65–68.

[61] Nigel Smart. *Cryptography Made Simple*. Springer, 2019.

[62] Rohini Poolat Parameswarath and Biplab Sikdar. “An Authentication Mechanism for Remote Keyless Entry Systems in Cars to Prevent Replay and RollJam Attacks”. In: *2022 IEEE Intelligent Vehicles Symposium (IV)*. 2022, pp. 1725–1730. DOI: 10.1109/IV51971.2022.9827256.

[63] Z. Shao et al. “Defending embedded systems against buffer overflow via hardware/software”. In: *19th Annual Computer Security Applications Conference, 2003. Proceedings.* 2003, pp. 352–361. DOI: 10.1109/CSAC.2003.1254340.

[64] Tanmaya Mishra, Thidapat Chantem, and Ryan Gerdes. “Survey of control-flow integrity techniques for real-time embedded systems”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 21.4 (2022), pp. 1–32.

[65] Sudeendra K Kumar et al. “A novel holistic security framework for in-field firmware updates”. In: *2018 IEEE International Symposium on Smart Electronic Systems (iSES)(Formerly iNiS)*. IEEE. 2018, pp. 261–264.

[66] Sebastian Vasile, David Oswald, and Tom Chothia. “Breaking all the things—A systematic survey of firmware extraction techniques for IoT devices”. In: *Smart Card Research and Advanced Applications: 17th International Conference, CARDIS 2018, Montpellier, France, November 12–14, 2018, Revised Selected Papers 17*. Springer. 2019, pp. 171–185.

[67] Shaza Zeitouni et al. “Atrium: Runtime attestation resilient under memory attacks”. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2017, pp. 384–391.

[68] Lorenzo Zuolo et al. “System interconnect extensions for fully transparent demand paging in low-cost MMU-less embedded systems”. In: *2013 International Symposium on System on Chip (SoC)*. 2013, pp. 1–6. DOI: 10.1109/ISSoC.2013.6675257.

[69] Arm Ltd. *TrustZone for Cortex-M*. https://www.arm.com/technologies/trustzone-for-cortex-m. Accessed: 2024-07-26. 2024.

[70] Joseph Yiu. *EW2017 - High-End Security Features for Low-End Microcontrollers*. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/ew2017---high-end-security-features-for-low-end-microcontrollers. Accessed: 2024-07-26. 2017.

[71] Victor Costan, Ilia Lebedev, Srinivas Devadas, et al. “Secure processors part I: background, taxonomy for secure enclaves and Intel SGX architecture”. In: *Foundations and Trends® in Electronic Design Automation* 11.1-2 (2017), pp. 1–248.

[72] Lionel Morel and Damien Couroussé. “Idols with Feet of Clay: On the Security of Bootloaders and Firmware Updaters for the IoT”. In: *2019 17th IEEE International New Circuits and Systems Conference (NEWCAS)*. 2019, pp. 1–4. DOI: 10.1109/NEWCAS44328.2019.8961216.

[73] Mario Noseda et al. “Performance analysis of secure elements for IoT”. In: *IoT* 3.1 (2021), pp. 1–28.

[74] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. 2st. Chapman and Hall/CRC, 2014. ISBN: 978-1-46657-026-9.

[75] Veena S Chakravarthi and Shivananda R Koteshwar. *System on Chip (SOC) Architecture: A Practical Approach*. Springer Nature, 2023. ISBN: 978-3031362415.

[76] Samuel Lindemer, Gustav Midéus, and Shahid Raza. “Real-time thread isolation and trusted execution on embedded RISC-V”. In: *Proceedings of the International Workshop on Secure RISC-V Architecture Design Exploration (SECRISC-V), Boston, USA, August 23* (2020).

[77] Alex Thomas et al. “Ertos: Enclaves in real-time operating systems”. In: *Woodstock ´18, New York, USA, June 03–05* (2018).

[78] Cesare Garlati and Sandro Pinto. “Secure IoT Firmware For RISC-V Processors”. In: *Embedded world, Nürnberg, Germany, March 01–05* (2021).

[79] Patrick Koeberl et al. “TrustLite: A security architecture for tiny embedded devices”. In: *Proceedings of the Ninth European Conference on Computer Systems*. 2014, pp. 1–14.

[80] Gary Mullen and Liam Meany. “Assessment of buffer overflow based attacks on an IoT operating system”. In: *2019 Global IoT Summit (GIoTS)*. IEEE. 2019, pp. 1–6.

[81] Wei Zhou et al. “Good motive but bad design: Why ARM MPU has become an outcast in embedded systems”. In: *arXiv preprint arXiv:1908.03638* (2019).

[82] CHIPS Alliance. *Rocket-Chip - Rocket Chip Generator*. Accessed: 2024-02-13. URL: `https://github.com/chipsalliance/rocket-chip`.

[83] Jerry Zhao et al. “SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine”. In: (May 2020).

[84] OpenHW Group. *OpenHW Group CORE-V CV32E40P RISC-V IP*. Accessed: 2024-02-13. URL: `https://github.com/openhwgroup/cv32e40p`.

[85] OpenHW Group. *OpenHW Group CORE-V CV32E40S RISC-V IP*. Accessed: 2024-02-13. URL: `https://github.com/openhwgroup/cv32e40s`.

[86] SpinalHDL. *VexRiscv - A FPGA friendly 32 bit RISC-V CPU implementation*. Accessed: 2024-02-13. URL: `https://github.com/SpinalHDL/VexRiscv`.

[87] YosysHQ. *PicoRV32 - A Size-Optimized RISC-V CPU*. Accessed: 2024-02-13. URL: `https://github.com/YosysHQ/picorv32`.

[88] OpenHW Group. *CV32E20 - Small 2-stage Pipeline 32 bit RISC-V Core*. Accessed: 2024-02-13. URL: `https://github.com/openhwgroup/cve2`.

[89] Keystone Enclave. *bootloader.c*. `https://github.com/keystone-enclave/keystone/blob/master/bootrom/bootloader.c`. Accessed: 2024-07-31. 2024.

[90] Julia Borghoff et al. “PRINCE–a low-latency block cipher for pervasive computing applications”. In: *Advances in Cryptology–ASIACRYPT 2012: 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings 18*. Springer. 2012, pp. 208–225.

[91] Jean-Philippe Aumasson and Daniel J Bernstein. “SipHash: a fast short-input PRF”. In: *International Conference on Cryptology in India*. Springer. 2012, pp. 489–508.

[92] Wikipedia contributors. *Block cipher mode of operation*. `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Electronic_codebook_(ECB)`. Accessed: 2024-08-08. 2024.

[93] Moses Liskov, Ronald L Rivest, and David Wagner. “Tweakable block ciphers”. In: *Advances in Cryptology—CRYPTO 2002: 22nd Annual International Cryptology Conference Santa Barbara, California, USA, August 18–22, 2002 Proceedings 22*. Springer. 2002, pp. 31–46.

[94] Hrushit Parikh et al. “Performance parameters of RTOSs; comparison of open source RTOSs and benchmarking techniques”. In: *2013 International Conference on Advances in Technology and Engineering (ICATE)*. 2013, pp. 1–6. DOI: `10.1109/ICAdTE.2013.6524742`.

[95] Tran Nguyen Bao Anh and Su-Lim Tan. “Real-Time Operating Systems for Small Microcontrollers”. In: *IEEE Micro* 29.5 (2009), pp. 30–45. DOI: `10.1109/MM.2009.86`.