UCLouvain

epl

École polytechnique de Louvain

Improving Security in Embedded Systems

Secure Multitasking in the ESP Privilege Separation Framework

Authors: **Jean-Christophe Bauduin, Lucas Roman**
Supervisor: **Pr. Ramin Sadre**
Readers: **François De Keersmaeker, Alexandre Vogel**
Academic year 2023–2024
Master [120] in Computer Science and Engineering
Master [120] in Computer Science

Abstract

This master thesis addresses the security challenges inherent in the proliferation of embedded devices by extending the ESP Privilege Separation Framework to support multi-user capabilities in real world applications. Our main contribution is the creation of a system that facilitates the execution of various user programs through the use of specially designed schedulers, specifically a round-robin scheduler and a preemptive scheduler with cooperative features. This multitasking capability allows device manufacturers to lease devices to multiple clients, facilitating the cohabitation of their applications on a single device, mirroring the practices that leading cloud providers have been employing for years.

Other enhancements include secure methods for handling sensitive data, efficient data-sharing pipelines between kernel and user space, and the integration of deep sleep functionality to preserve battery life. The new capabilities are benchmarked against the native ESP IDF framework. The results demonstrate that, although the new features introduce some overhead, careful application design can mitigate these impacts, rendering the overhead negligible in typical use cases.

Source Code

All the implementation code, experimental data, and plots for this thesis can be accessed in our GitHub repository at
`https://github.com/jbd0101/esp32s3-privilege-separation`

Acknowledgments

We would like to express our deepest gratitude to Professor Ramin Sadre, whose guidance, expertise, and continuous support have been invaluable throughout the development of this thesis. Your insightful feedback and encouragement have significantly contributed to the quality and direction of our research.

We are also immensely grateful to François De Keersmaeker for your detailed reviews and constructive critiques. Your perspectives have been instrumental in refining our work and pushing the boundaries of our understanding.

Additionally, we would like to extend our sincere thanks to Alexandre Vogel, who will join Pr. Ramin Sadre and François De Keersmaeker in evaluating this thesis. Your time and effort in assessing our work are greatly appreciated.

To our friends, family, and loved ones, your unwavering support and encouragement have been the foundation upon which we have built this thesis. Your belief in us has provided the motivation and strength needed to persevere through the challenges of this journey.

Thank you all for your contributions, which have made this achievement possible.

Use of AI tools

In the course of writing this document, we utilized various language models, including Chat-GPT. Our primary purpose for employing these models was not content generation, but rather to enhance the scientific accuracy and clarity of our own writing. Additionally, during the implementation phase, we leveraged the Microsoft Copilot plugin to expedite code writing, while ensuring that the conceptual foundations behind our design decisions remained entirely our own. Overall, our approach adheres to the guidelines set forth by the EPL regarding the responsible use of artificial intelligence tools.

Contents

Glossary

ABI Application Binary Interface. 17

DCACHE Data Cache. 28

DRAM Dynamic RAM. 27, 62

eFuse Electronic Fuse. 12

GPIO General Purpose Input/Output. 9, 12, 14, 24, 32

GPOS General Purpose Operating System. 5

HAL Hardware Abstraction Layer. 14

I2C Inter-Integrated Circuit. 14, 15

ICACHE Instruction Cache. 28

IDF IoT Development Framework. 9, 11–14, 16, 18, 28, 29, 31, 38–42, 44, 46, 49, 52, 55, 57, 62, 65

IoT Internet of Things. 1–3, 5, 8, 12, 24

IPC Inter-Process Communication. 12

IRAM Instructions RAM. 27, 28, 62

MCU Microcontroller Unit. 9

MMU Memory Management Unit. 5–7, 11

MPU Memory Protection Unit. 5–7

NVS Non Volatile Storage. 26, 28–30, 60

OTA Over-The-Air. 12, 14, 17, 18, 29, 34, 61, 62

PII Personally Identifiable Information. 29

PMS PerMiSsion control. 25, 28, 32, 59

PSF Privilege Separation Framework. 14, 19, 39–42, 45–47, 49, 52, 55–58

PSRAM Pseudo-Static RAM. 27, 28, 60, 61

RF Radio Frequency. 28

ROM Read-Only Memory. 9

RTC Real-Time Clock. 14, 24, 53

RTOS Real Time Operating System. 5, 13

SDK Software Development Kit. 8

SMC Secure Monitor Call. 4

SoC System-on-Chip. 3, 6–9, 12

SPI Serial Peripheral Interface. 14

SRAM Static RAM. 9, 27, 28

TEE Trusted Execution Environment. 4

VM Virtual Machine. 5

Chapter 1

Introduction

In the USENIX 2023 conference held in August 2023, P. de Vaera and A. Perrig introduced Kimya, a novel approach aimed at safeguarding smart speaker users from unauthorized eavesdropping [1]. They initiated their research with the premise that the existing privacy measures of smart speakers are inadequate. The opacity of the LED indicators, for instance, enables the smart speaker to be in a perpetual listening mode without the user's knowledge. Furthermore, they pointed out that in the event of a smart speaker being compromised, there are no existing safeguards to prevent a hacker from gaining unrestricted access to the microphone. This highlighted the urgent need for more robust security measures in smart speaker technology.

Their proposal involves placing the code responsible for managing critical functionalities such as microphone input, wake word detection, and LED indicators within the secure enclave provided by TrustZone. This architectural decision allows them to furnish users with assurances that the LED indicators accurately reflects the current status of the microphone, and that no rogue code is clandestinely recording audio without user consent.

The integration of TrustZone technology on Cortex-M chips presents a compelling solution to the security challenges inherent in smart speakers, particularly concerning privacy-sensitive features like microphone usage. By isolating critical components within a secure enclave, De Vaere and Perrig's approach effectively mitigates the risk of unauthorized access or tampering with sensitive functionalities.

In the course of their investigation, the researchers concentrated on smart speakers powered by an electrical outlet and equipped with robust ARM chips. This sparked our curiosity about the present condition of the low-power Internet of Things (IoT) landscape. As a result, we began by examining some widely used IoT devices, such

as Tuya compatible sensors, and gained several insights: Tuya allows any company to manufacture their own sensors, link them to their network, and market them. Other devices, such as detectors from LinknLink, allow users to install their own firmware on the device. However, we were unable to locate IoT devices that offer solid assurances of sensor isolation and protection.

Drawing a parallel to Kimya, could a company potentially market a secure code that securely manages data acquisition and indicators, while allowing other companies to embed their processing code in it on inexpensive microcontrollers?

Previous research, as proposed by [2], aims to address this challenge through the utilization of `femto-containers`, which essentially function as interpreters for user code. While this approach effectively achieves adequate isolation, it comes with the drawback of performance limitations compared to native C execution, as programs are bound to the interpreter. For instance, leveraging specialized instructions like SIMD on an ESP32-S3 would necessitate updating the interpreter, introducing potential compatibility issues and additional overhead.

In an attempt to address this concern, we decided to explore the possibility to use the "Two worlds" controller of the ESP32-S3 to create a solution that would permit to have

1. A separate secure zone that can access the peripherals, handling the sensitive information.
2. One or more user application that can handle the processing and sending of the gathered data.

The task at hand is to enable weak and low-power devices to offer security comparable to the TrustZone on ARM chips.

1.1 Structure of the Thesis

The remainder of our thesis is organized as follows: Chapter 2 starts by exploring the current studies on the subject of secure computing and virtualization in embedded devices. Subsequently, Chapter 3 starts by providing the necessary background knowledge to ensure a clear understanding of the technical aspects of our work. Next, Chapter 4 outlines our contributions and explain the design choices we have made. Chapter 5 then assess the effectiveness of our contributions by comparing them to other implementations and discussing any performance implications. Finally, Chapter 6 addresses the limitations of our work and suggest possible future research directions, and Chapter 7 concludes with a summary of our findings.

Chapter 2

State-of-the-Art

In this chapter, we explore current research related to trusted execution and virtualization within the context of embedded computing. Specifically, we delve into the ARM TrustZone security extension—a hardware component designed to enhance security, particularly well-suited for more powerful System-on-Chip (SoC) boards. Additionally, we discuss a software-based security solution that accommodates a broader range of SoC boards, including those with more limited capabilities.

2.1 ARM TrustZone

The ARM TrustZone is a hardware security extension that was introduced into ARM processors in 2004 [3]. This extension introduces the concepts of secure world and normal world, where the code executed by the processor runs in either domain. Initially, the secure context switching between the two worlds was handled by the so-called secure monitor, which is a privileged piece of software. The Cortex-M processors excludes the need of secure monitor and relies on new specific instructions to switch from one world to the other.It improves the power consumption as well as the overall speed of the microcontroller.

It is worth noting the existence of a significant competitor, Intel SGX, which is another widely recognized security solution. However, this literature review deliberately focuses on ARM TrustZone for several reasons. Firstly, Intel SGX primarily targets server platforms rather than IoT devices, which are the main interest of our research. Secondly, while Intel SGX provides secure code enclaves that remain protected even from the operating system kernel, our research objectives do not require this level of isolation. Instead, ARM TrustZone offers a security

model better suited to the needs and constraints of IoT applications, aligning more closely with the goals of this study.

2.1.1 Trusted Execution Environment

In [4], Sabt et al. propose a definition for Trusted Execution Environment (TEE). *TEE is a tamper-resistant processing environment that runs on a separation kernel. It guarantees the authenticity of the executed code, the integrity of the runtime states (e.g., CPU registers, memory, and sensitive I/O), and the confidentiality of its code, data and runtime states stored on a persistent memory. In addition, it shall be able to provide remote attestation that proves its trustworthiness for third-parties. The content of TEE is not static; it can be securely updated. The TEE resists against all software attacks as well as the physical attacks performed on the main memory of the system. Attacks performed by exploiting backdoor security flaws are not possible*

TrustZone constitutes a natural enabler for building TEE support systems. Essentially, the secure world provides a restricted execution environment where the TEE can reside. Whenever the system boots, the processor enters the secure world to give to any privileged firmware the chance to set up its internal data structures, configure the interrupt controller of the entire system, and set up protections for secure memory regions and peripherals [5]. Once this initial setup is completed, the processor switches to the normal world and transfers control to the OS bootloader. Since the OS operates in the normal world, it does not have the privileges to access the secure world's memory or configure interrupts to gain access to it. The solution to enable the normal world to access some parts of the secure world is to execute a Secure Monitor Call (SMC) which causes a SMC exception, which in turn causes the processor to enter the so-called Monitor mode [6]. The monitor mode is completely different from other supported modes, because when the processor runs in this mode, the state is always considered secure[7]. These mechanisms facilitate the execution of a trusted program within the secure world without relying on the integrity of the operating system in the normal world. If the operating system is compromised, any attempts to access the secure world's address space will trigger violations, leading to exceptions being thrown.

2.1.2 Virtualization

Hardware virtualization is a technology that allows a single physical machine to run multiple isolated operating systems simultaneously. This is achieved by abstracting the physical hardware components of the host machine into multiple sets of virtual hardware components, each of which can be independently controlled by a guest

operating system.

While TrustZone technology is primarily designed for security applications, it also facilitates a form of system virtualization that is enabled by the hardware extension. With a virtual hardware support for dual world execution, as well as other TrustZone features like memory segmentation, it becomes feasible to establish temporal and spatial segregation between different execution environments. Essentially, non-secure software operates within a Virtual Machine (VM), the resources of which are entirely managed and overseen by a hypervisor running in the secure world [5].

Several researchers have developed various types of virtualization and hypervisors for different purposes. In this study, we have selected two specific types: dual-guest and multi-guest virtualization. We will briefly introduce their design and objectives.

The traditional configuration of dual-guest operating systems is the following: the secure world hosts the Real Time Operating System (RTOS) (e.g. FreeRTOS) and the hypervisor, while the General Purpose Operating System (GPOS) (e.g. Linux) is allocated to the normal world. Both guest operating systems utilize the same CPU. However, the principle of asymmetric design stipulates that the GPOS is only scheduled when the RTOS is idle, while simultaneously ensuring that the RTOS can interrupt the execution of the GPOS [8].

The architecture of multi-guest virtualization, as introduced by Pinto et al. [9]. It operates as follows: the hypervisor functions in the most privileged mode within the secure world, specifically, the monitor mode. Its primary responsibilities include managing the virtual CPUs, ensuring temporal and spatial isolation, and mediating interrupts designated to inactive guests. The guest operating systems can be encapsulated within the secure and non-secure world sides: the active guest partition operates within the non-secure world sides, while the states of other inactive guests are preserved within the secure world side.

As a side note, Intel proposes SGX enclave as a security feature on Intel Xeon processors, however it is quite different to ARM TrustZone. First, The intel Xeon processors are made for servers and not IoT. Second, Intel SGX gives access to secure enclaves that are protected even from the kernel of the OS.

2.2 MPU and MMU based Solutions

In the next paragraph, papers will use either the Memory Management Unit (MMU) or Memory Protection Unit (MPU). The MPU is a hardware component located in the chip that verifies the authorization to a memory address. The MMU is a superset of the MPU: it handles the permissions, the virtual addresses and the

page tables.

Prior research has proposed lightweight solutions that utilize the MPU. In 2018, a study [10] employed the MPU of an inexpensive microcontroller, specifically the Texas Instruments MSP430FR5969. This chip is a high-efficiency, low-power microcontroller with basic capabilities, lacking features such as Wi-Fi and Bluetooth, and operating at a low clock speed.

The researchers developed a solution, named AmuletOS, which is primarily software-based. The compiler is required to insert code around the functions to facilitate the switching between "worlds". As outlined in their paper, this approach allows for multiple user applications that cannot interfere with the protected environment, eliminating the need for any hardware support.

The paper, however, acknowledges certain limitations : the solution requires the compiler to add surrounding code, and it is susceptible to user applications jumping to kernel code through methods such as "goto" or pointer manipulation

In 2017, another study [11] introduces an innovative solution leveraging the Memory Management Unit (MMU). This research led to the creation of a new operating system named Tock OS. Tock OS, written in Rust, is designed with an integrated approach, incorporating both the kernel and user applications from the inception of the project.

Tock OS categorizes user applications into two types:

- Capsules, which are tasks executed with preemptive scheduling within the kernel.
- Processes, which reside in the user space and are equipped with memory protection.

The kernel can be accessed by the user application through the system call interface. This design allows efficient communication and control between the user applications and the kernel.

2.3 Conclusion

In summary, two prevailing trends emerge in the field of System-on-Chip (SoC) security. Firstly, for larger and more powerful SoCs, ARM TrustZone stands out as the most robust and secure enclosure, providing strong guarantees. This hardware-based security extension isolates secure and non-secure worlds within the same processor, ensuring that critical operations and sensitive data remain protected.

However, it places the responsibility on the developer's shoulders to construct a comprehensive security framework around it.

On the other hand, alternative solutions based on MPUs and MMUs also exist. These components, although lacking the dedicated hardware isolation of TrustZone, play a crucial role in enforcing security policies. Interestingly, neither of these MPU and MMU solutions relies on dedicated hardware components to safeguard the system. Instead, they both rely on software-based guards. Developers configure access permissions, define memory regions, and enforce runtime checks through software routines. While this approach lacks the inherent security guarantees of TrustZone, it offers flexibility and compatibility across a wide range of SoCs.

Chapter 3

Background Knowledge

This chapter's purpose is to equip readers with the necessary background knowledge to fully grasp the context of our work. It is assumed that readers have a basic understanding of computer system architecture, security, and networking. The concepts discussed here can be complex, therefore, we encourage readers to refer to the provided references, including documentation and scientific papers, should they have any lingering questions.

3.1 ESP System-on-Chip

The ESP System-on-Chip (SoC) is a series of integrated circuits designed specifically for IoT applications, developed by Espressif Systems [1], a leading Chinese semiconductor manufacturing company. Espressif's ESP SoCs are known for their low cost, low power consumption, and integrated Wi-Fi connectivity, making them popular choices for a wide range of IoT devices such as smart home products, wearables, and industrial automation.

ESP SoCs are supported by an active developer community and comprehensive Software Development Kit (SDK) that simplify the process of programming and deploying IoT applications. They are widely used due to their affordability, versatility, and ease of use, enabling developers to create connected devices with minimal hardware complexity and power consumption.

ESP SoCs are categorized into six distinct series, each designed with specific capabilities and power profiles optimized for different applications. Users must assess their requirements to identify the most suitable option within these series.

[1] `https://www.espressif.com/en/products/socs`

Throughout this thesis, the ESP32-S3 [2] model was consistently utilized. This model features a dual-core Xtensa 32-bit LX7 processor, accompanied by 512 KB of Static RAM (SRAM) and 384 KB of Read-Only Memory (ROM), integrated within the chip. Notably, the primary motivation for selecting this model, aside from its affordability and widespread availability, stemmed from its incorporation of the World Controller peripheral—a feature that significantly influenced our choice of SoC for experimentation and analysis.

3.1.1 Development Board

A development board is used during the development phase of creating a new embedded device. It allows developers to familiarize themselves with a microprocessor (in this case, the ESP32-S3) without needing to customize it, and it comes equipped with additional peripherals and General Purpose Input/Output (GPIO) pins. This enables the creation of a prototype using a general-purpose board. However, this approach has certain drawbacks. Since the board is designed for general use, it lacks specific optimizations, and the additional components can affect power consumption. The final design, which includes only the necessary components to minimize cost and optimize power usage down to the last milliampere, is typically achieved at the end of a commercial or research project.

Throughout this thesis, we consistently use the ESP32 Development Kit C1 [3] that features the with the ESP32-S3-WROOM-1, a powerful, generic Wi-Fi and Bluetooth Low Energy MCU module that has a dual-core CPU, a rich set of peripherals, and provides acceleration for neural network computing and signal processing workloads. This development board is shown in Figure 3.1.

3.2 FreeRTOS

FreeRTOS is an open-source real-time operating system kernel designed for embedded devices, known for its small size and simplicity. It has gained popularity across various microcontroller platforms due to its efficient task management, ease of integration, and extensive feature set. Espressif Systems, the creators of the ESP32 series of microcontrollers, have incorporated FreeRTOS into their IoT Development Framework (IDF) to take advantage of these features while also adding custom API functions tailored to their hardware. Notably, FreeRTOS was originally developed for single-core CPUs, while the ESP32 features a dual-core CPU. To address

[2] `https://www.espressif.com/en/products/socs/esp32-s3`

[3] `https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/hw-reference/esp32s3/user-guide-devkitc-1.html`

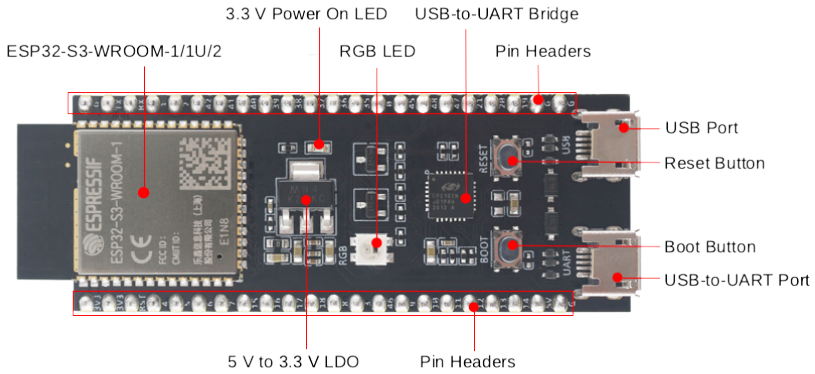

Figure 3.1: ESP32-S3 DevKit C1 annotated picture [12]

this issue, Espressif extends the FreeRTOS API to manage tasks in a dual-core environment, including for example the ability to pin tasks to a specific core using the `xTaskCreatePinnedToCore` [4] function.

3.2.1 Task Management

FreeRTOS provides preemptive multitasking capabilities, which means that multiple tasks can run seemingly simultaneously by sharing the CPU time. This is crucial for embedded systems like those using ESP32, as it allows handling various operations concurrently, such as sensor data processing, communication, and user interface management.

Tasks in FreeRTOS are analogous to threads in traditional operating systems. Each task is defined by a function and runs independently. Tasks can be created using the `xTaskCreate` [5] function, which takes parameters such as the task function, name, stack size, priority, etc.

Tasks can have different priorities, allowing more critical tasks to preempt less critical ones. This ensures that important operations are given more CPU time. Priorities are assigned when tasks are created and can be changed dynamically.

[4] `https://docs.espressif.com/projects/esp-idf/en/v4.3/esp32/api-reference/system/freertos.html`

[5] `https://www.freertos.org/a00125.html`

3.2.2 Inter Task Communication

Among other available solutions to transfer data from task to task, task notifications are a lightweight, efficient way to send data to tasks and synchronize events. They behave as binary semaphores or counting semaphores but more memory-efficient and faster [6]. Task notifications are sent using functions such as `xTaskNotify` [7] and received with `xTaskNotifyWait` [8].

Task notifications can carry an integer value, providing a mechanism for sending data or status flags. This makes them suitable for simple inter-task communication without the overhead of more complex data structures.

3.2.3 Additional Features

FreeRTOS offers additional features essential for efficient task management. It includes message queues for thread-safe communication between tasks, binary and counting semaphores for resource access and synchronization, and mutexes for managing shared resources. Software timers enable tasks to execute at specified intervals, useful for monitoring and periodic actions. Event groups allow tasks to handle multiple events simultaneously by waiting for combinations of event flags. Additionally, FreeRTOS provides dynamic memory allocation schemes to optimize RAM usage and balance fragmentation and performance, all of which are integrated into the ESP IDF.

3.3 Process isolation in Traditional Computing

Process isolation is a foundational principle in most operating systems, ensuring both the overall system's integrity and the integrity of individual processes. It involves separating each process from others to prevent unauthorized access to sensitive data such as passwords or secrets. Additionally, process isolation enhances system resilience by allowing individual processes to fail without crashing the entire machine.

In modern operating systems, process isolation is primarily achieved through virtual address spaces. Each process perceives itself as having exclusive access to the entire memory, although it is actually the Memory Management Unit (MMU) that allocates specific portions of physical memory to each process. The MMU also ensures that processes cannot access physical memory outside their designated

[6] `https://www.freertos.org/2020/09/decrease-ram-footprint-and-accelerate-execution-with-freertos-notifications.html`

[7] `https://www.freertos.org/xTaskNotify.html`

[8] `https://www.freertos.org/xTaskNotifyWait.html`

address spaces. This approach not only strengthens security but also simplifies application development and improves efficiency.

In certain scenarios, there arises a necessity to facilitate data sharing among multiple processes. This requirement is addressed through the utilization of Inter-Process Communication (IPC) channels. IPC mechanisms like shared memory, local sockets, and internet sockets serve distinct purposes and have specific characteristics. Shared memory IPC involves processes sharing a region of memory, allowing them to read and write data directly. This method is efficient for high-speed communication between processes on the same system but requires synchronization mechanisms to manage concurrent access. Local sockets, such as Unix domain sockets, facilitate communication between processes on the same host using file-like interfaces. They are reliable and offer features like stream or datagram-based communication, ideal for inter-process communication within a single machine. On the other hand, internet sockets, like TCP/IP or UDP/IP sockets, enable communication between processes running on different hosts over a network. These sockets use networking protocols to transmit data over the internet or local network, providing inter-process communication across diverse systems.

3.4 IoT Development Framework

The Espressif IoT Development Framework (IDF) [9] serves as the conventional development tool for developers that want to program IoT applications using ESP chips; supplying an abstraction layer over the API of the ESP32, it integrates an adapted version of FreeRTOS and supplies all the build tools necessary to flash and configure the ESP. Typically, user code is structured as a monolithic application within ESP IDF, using its API to interact with hardware components like sensors, GPIO pins and using their framework in order to integrate Over-The-Air (OTA) updates easily. For the production units, ESP provides scripts to flash, encrypt the flash and burn the needed Electronic Fuse (eFuse) of the SoC described hereafter.

An eFuse is a hardware component consisting of a single-bit field programmable to a value of 1 by the user, with no possibility of reversal to 0. This eFuse incorporates a silicided polysilicon fuse, which is altered via electromigration [13]. Certain hardware modules, like cryptographic modules, directly access eFuse data through the eFuse API on the ESP platform to adjust configuration settings. In ESP32 chips, there are four 256-bit eFuse blocks available for programming, with some reserved by the system.

[9] `https://docs.espressif.com/projects/esp-idf/en/stable/esp32/index.html`

3.5 Privilege Separation Framework

In this thesis, we adopt the Privilege Separation Framework [10], an ongoing project by Espressif Systems, currently in beta. This framework aims to introduce a new layer of abstraction above ESP IDF, separating traditional monolithic Real Time Operating System (RTOS) firmware into two independent executables: a protected application and a user application, with a well-defined system-call interface between them. The protected application operates at higher privilege, accessing full system memory and peripherals, while the user application has restricted memory and peripheral access, governed and granted by the protected application.

At build time, two distinct, independent firmware binaries are created. The protected application comprises the operating system, networking stack, and device drivers, performing the majority of tasks. The user application is minimal, mainly handling business logic and utilizing services through a system call-based interface provided by the protected application. The protected application has runtime capabilities to configure memory space and peripheral access as needed. Importantly, any exceptions occurring within the user application do not impact the functionality of the protected application: the exception will trigger an interrupt that will be caught in the protected environment. Figures 3.2 and 3.3 depict the differences in architecture between both frameworks.

[10] `https://docs.espressif.com/projects/esp-privilege-separation/en/latest/esp32c3/`

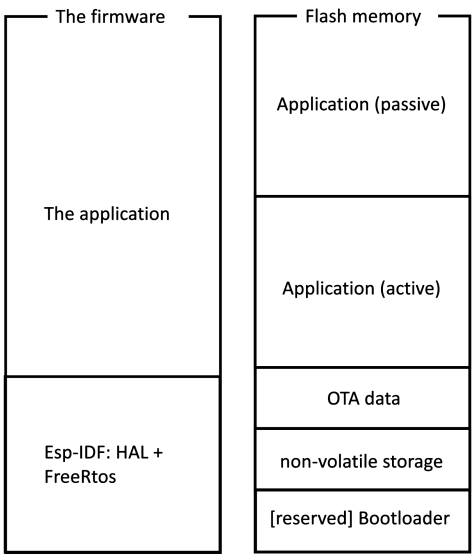

Figure 3.2: Typical structure of the firmware and flash memory

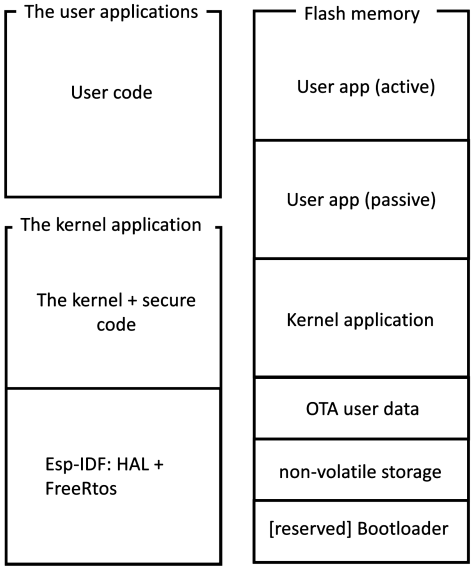

Figure 3.3: Structure with equivalent capabilities using the world controller

Figure 3.4 illustrates the architecture of the entire application, highlighting the different layers of abstraction and frameworks. The layers are described from the top down as follows:

1. The highest layer contains the non-secure world code, also known as the user code, which manages the application business logic, and the secure world code, which handles sensitive data.
2. The Privilege Separation Framework (PSF) configures the world registers, loads both secure and non-secure applications, and manages flash encryption and Over-The-Air (OTA) updates.
3. Espressif's extended FreeRTOS supports multicore capabilities and manages the scheduler, mutexes, software timers, and tasks.
4. The Hardware Abstraction Layer (HAL) manages the configuration of General Purpose Input/Output (GPIO) pins, Serial Peripheral Interface (SPI), hardware timers, and other peripherals.
5. The lowest layer deals with the most fundamental hardware configurations, including cores and peripherals such as Inter-Integrated Circuit (I2C), Real-Time Clock (RTC), and the world controller.

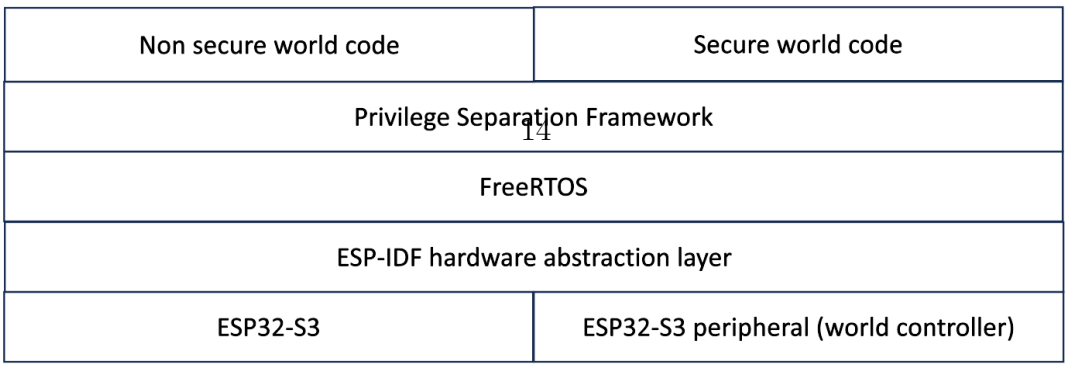

Figure 3.4: ESP IDF layers

3.5.1 Two worlds

To implement this privilege separation, the framework employs a so-called two-world concept: `WORLD0`, representing the protected space, also called kernel space or secure world, and `WORLD1`, representing the user space.

In the secure world, access is restricted based on memory addresses. All memory addresses associated with peripherals and the non-secure world are blocked. Consequently, any user code that needs to interact with peripherals such as Crypto or I2C must do so via the kernel. Figure 3.5 illustrates this behavior. Furthermore, any functions that require access to these registers (e.g., `xTaskCreate` from FreeRTOS or MbedTLS) must also operate through the kernel. However, purely software-based libraries, such as the JSON parser, can be included without modification.

Non-critical components can be integrated into the protected application with implemented system calls. Here, the protected application leverages the component's APIs and incorporates the necessary system calls for its functionality. When a user application accesses this component's API, it triggers the corresponding system call. Alternatively, dual instances of a component can coexist, one in each world, where the secure world possesses its own instance while the user world has a separate one. For example, the `heap` component operates this way, with independent heap allocator instances in both worlds, oblivious to each other's existence.

Common high-level ESP libraries-like the HTTP(S) clients-can be imported in the user code if the function that they are using in background is available from the system call interface.

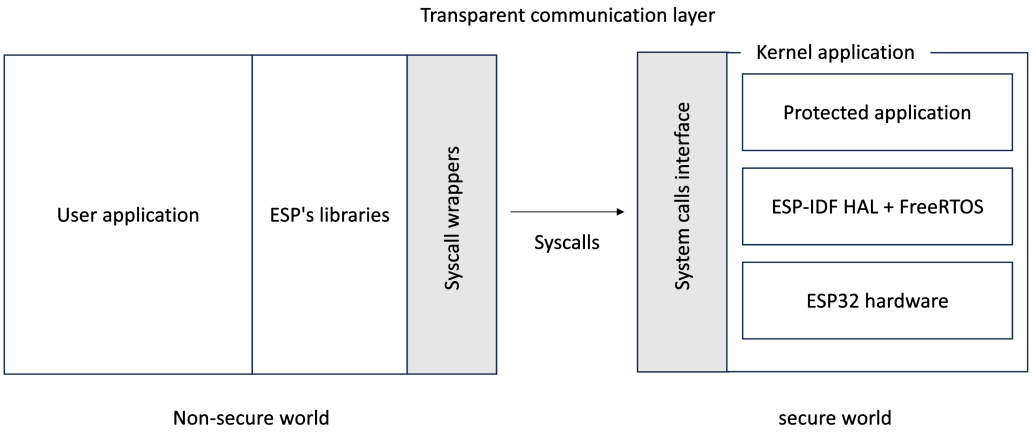

Figure 3.5: Communication layer between worlds

3.5.2 Boot Up Process

The application's boot-up process mirrors that of ESP IDF. The entry point is the `app_main` function within the secure world. The protected application is responsible for setting permissions for the user application, configuring memory regions, and loading the user code from flash. Additionally, it can expose a collection of system calls that the user world can utilize to request access to specific resources.

The user application relies on permissions granted by the protected application. When the user application requires certain privileged operations, it must invoke the designated system calls provided by the protected application to accomplish them.

3.5.3 System Calls

To seamlessly transition from user space to protected space, the `ecall` instruction is used to generate a synchronous exception. There are two types of system calls: those related to the native API, and custom system calls created by developers. With privilege separation, an ecall will initially trigger a context switch to the kernel by generating an interrupt to the `syscall_handler`. Subsequently, a second handler, which is not an interrupt, will invoke the corresponding function. Each function in the kernel is identified by a number, which is stored in the `a2` register during the context switch. the 6 other registers are used to pass parameters and to return the value.

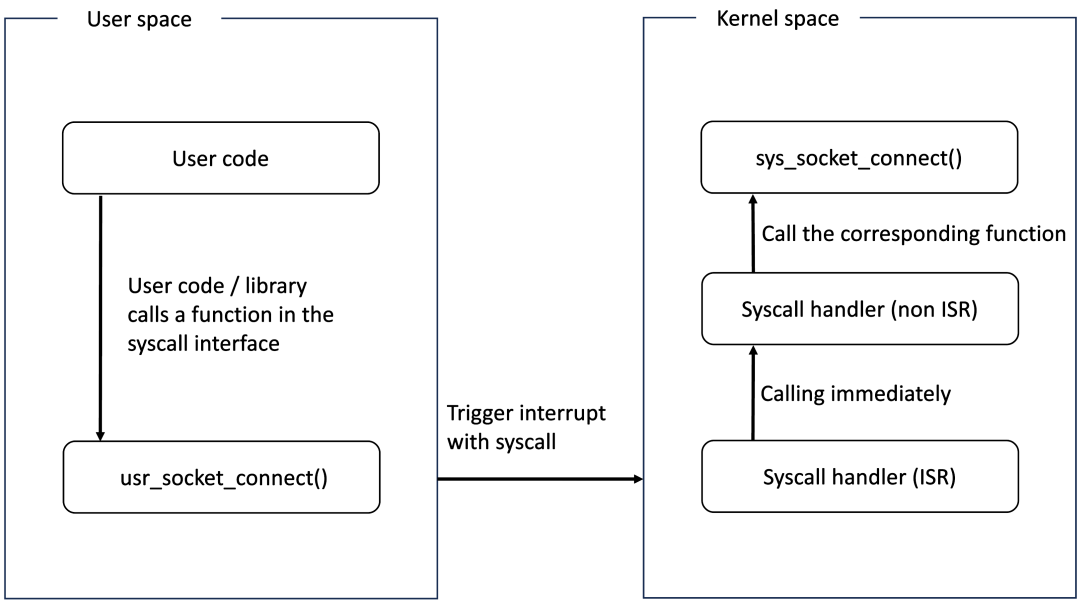

Figure 3.6: System call architecture

For core components invoked from the user space, system calls are used to execute them in the protected space. Importantly, the user application does not need to define different API prototypes for these components.

When using an API function called "NativeFunction", behind the scenes, the system needs to execute the corresponding system call, which would be "usr_NativeFunction". This translation is done automatically during the building of the user application. The build process parses the `syscall.tbl` file (which contains all the system calls) and searches for all the system calls that use the "common" Application Binary Interface (ABI). The `defsym` option of the linker is then used to map all references of `<symbol>` to the address of `<usr_symbol>`. As a result, when the user application is built, all instances of `<xNativeFunction>` are effectively replaced by `<usr_xNativeFunction>`.

When creating custom system calls, developers must select a new system call number and add an entry in a custom system call table (`.tbl` file), which is automatically parsed during application building. Next, developers need to implement a wrapper function that invokes `ecall` with the assigned system call number. Finally, the developer must implement the actual function for the system call, which executes in the protected space upon invocation of the syscall request with the corresponding system call number.

3.5.4 Over-The-Air (OTA) Updates

OTA updates represent a method by which software updates are seamlessly delivered to devices remotely by developers, eliminating the need for manual download and reinstallation by users. On ESP devices, these updates can be facilitated through either Wi-Fi or Bluetooth, with the latter being less commonly utilized.

Within the framework of privilege separation, it becomes feasible to independently update the user application and the protected application. This approach allows for more lightweight updates.

To implement OTA updates, the partition tables must be modified to include a new passive partitions for each world you want to enable the OTA on. Additionally, two 4 Kb partitions are reserved for selecting the active partition for each application realm. Given the critical nature of this feature, OTA updates are managed by the secure world, which is responsible for handling each update. The user world can initiate an update via a system call, with the secure world then validating the provided URL from the user application and scheduling the update task.

The update task involves downloading the new firmware, storing it in the designated passive partition, updating the selection partition, and initiating a device reboot.

Developers should be mindful that a reboot will result in the loss of all volatile data.

To bolster the security of OTA updates, firmware verification can be implemented by appending a signature to the binary. During the update process, the signature is verified, and the update is aborted if verification fails.

As in ESP IDF, a rollback mechanism is in place to address update failures. In the event of a failure, the rollback mechanism readjusts the selection partition entry, allowing the device to reboot without entering a failed boot loop.

Chapter 4

Real World Multi-User Framework

The main goal of this research is to enhance the ESP Privilege Separation Framework (PSF) by introducing multitasking capabilities while maintaining strict task isolation. During the implementation of this feature, we acknowledged the critical need to empower users to manage sensitive information securely, safeguarding against unauthorized access and physical tampering. Consequently, we devise a method to securely store user secrets and prevent unauthorized tasks from accessing information that does not belong to them. We also recognize the significance of establishing an efficient data transfer mechanism between secure and user environments. Leveraging insights from networking and computer system architectures, we adopt the message queuing principle and implement a queue-based data transmission system, which we refer to as the pipeline. These contributions form the focus of our work, and this chapter provides a detailed explanation of their implementation and design considerations.

4.1 Secure Multitasking

To enable the execution of their tasks within the system, end users are required to submit their task's source code to the device owner. This practice, while necessary, introduces a notable drawback as it potentially exposes proprietary code to untrusted entities, a concern which is thoroughly examined in the discussion chapter (see Chapter 6). Subsequently, upon integration of the code into the code base, the device owner must undergo the process of rebuilding and flashing the entire application onto the device.

During the boot-up sequence, after the initialization of the user environment by the kernel, a kernel task is invoked. This task serves the purpose of allocating the

requisite memory on the kernel's heap and determining the quantity of tasks to be started. The allocated memory encompasses a contiguous memory area, the size of which corresponds to the cumulative size of the user stacks, along with an array of task contexts. These task contexts include pointers to task stacks and their respective sizes. This memory allocation is described in Figure 4.5. Alternatively, their structure definition can be found in Code Snippet 4.1 The decision to utilize a contiguous array for stack memory facilitates efficient manipulation and iteration through task stacks. However, a current limitation lies in the complete copying of each stack to the buffer, irrespective of whether only a fraction of it is utilized. This optimization was not implemented in our framework, but its viability is discussed in the discussion chapter (see Section 6.2).

Code Snippet 4.1: User task context structure

```
typedef struct {
    void *stack;
    int stack_size;
    void *task_errno;
    void *task_handle;
} usr_task_ctx_t;
```

Subsequently, control returns to the dispatcher task within the user environment. This task is responsible for sequentially initiating all user tasks and immediately suspending them. This preemptive suspension ensures that while tasks are started in the user environment, they are not permitted to execute until directed by the kernel dispatcher. Following the suspension of each task, the user dispatcher transmits the task context and stack content to the kernel for storage within the designated buffers.

Upon completion of these tasks, the user world dispatcher concludes its operation and initiates the kernel dispatcher. The kernel dispatcher functions as a bespoke round-robin task scheduler, entering an infinite loop wherein it cycles through user tasks. Each task is allocated a predetermined time range, defaulting to 2000 ms. We arbitrarily chose this value, and it could be changed according to any other use case. The time range is however required to be shorter than the watchdog, otherwise the device will reboot. Once the time range is exceeded, the task is suspended, and its stack and context are stored in their respective buffers. Subsequently, the next task on the list is resumed for execution. Figure 4.1 illustrates the workflow of the scheduling operation.

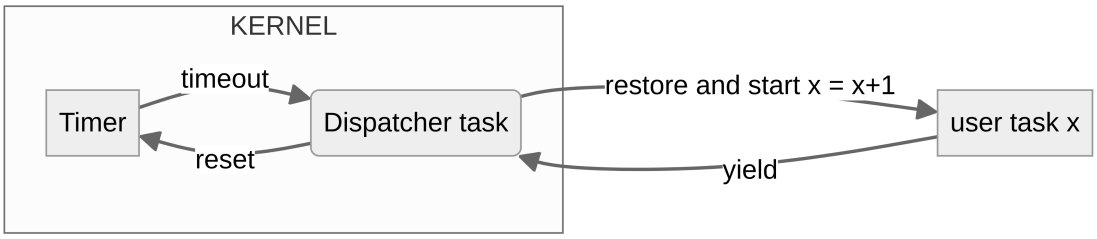

Figure 4.1: Round-robin scheduler with timer

4.1.1 Preemptive Scheduler with Yielding Function

Upon closer examination, it is evident that many user applications do not necessitate the full 2000 ms for their operations. Most tasks are quite straightforward, involving actions such as reading sensor data, performing basic calculations, and transmitting or storing results. Consequently, waiting until the entire allocated time elapsed before handing control back to the scheduler seemed inefficient. With this realization, we set out to develop an additional scheduler layer that allows tasks to yield, on top of the preemptive scheduling provided by the round-robin implementation.

The core idea is simple: tasks have the liberty to pause their execution at any point, whether due to waiting for resources, encountering logical barriers, or simply completing their current task with no immediate follow-up actions. When a task pauses, the scheduler steps in to preserve its context, copy its stack to a designated buffer, and move on to execute the next task. However, to prevent the risk of task starvation, we opted for a hybrid approach, combining both cooperative and preemptive scheduling. Thus, if a task exceeds its allocated time without voluntarily giving back control, the scheduler intervenes, though still ensuring the preservation of the task's context and stack, and then proceeds with the next task.

To ensure that this optimization does not compromise the overall system performance, we embarked on exploring resource management techniques that do not rely on semaphores or mutexes. Eventually, we settled on utilizing task notifications, which, according to the FreeRTOS documentation, executed 45% faster than semaphores while consuming less RAM [1].

In FreeRTOS, upon task creation, an array of task notifications is allocated. Each notification within the array features a binary state indicating either *"pending"* or *"not pending"*, along with a 32-bit notification value.

[1] `https://www.freertos.org/2020/09/decrease-ram-footprint-and-accelerate-execution-with-freertos-notifications.html`

Triggering a task notification transitions the state of the targeted notification to *"pending"*. Much like a task can block while waiting for a semaphore to become available, it can also block while awaiting a task notification to transition to the *"pending"* state. Although it is possible to modify the task notification value, such functionality was not necessary in our case, as we aimed to replicate the behavior of a binary non-counting semaphore.

We introduce yet another custom system call to implement this feature. Despite the overhead associated with it, we find it necessary, as the dispatch task resides in the kernel world while user tasks naturally operate in the user world. During the dispatcher task initialization, its task handle is stored in a global variable, which the `notify_dispatch` function accesses. When a user intends to yield, it executes the newly created system call, that triggers the `notify_dispatch` function, thereby notifying the dispatch task and facilitating an expedited handover, preempting the initially designated 2000 ms limit. Figure 4.2 shows the workflow of the device when using the new scheduler to permit task yielding.

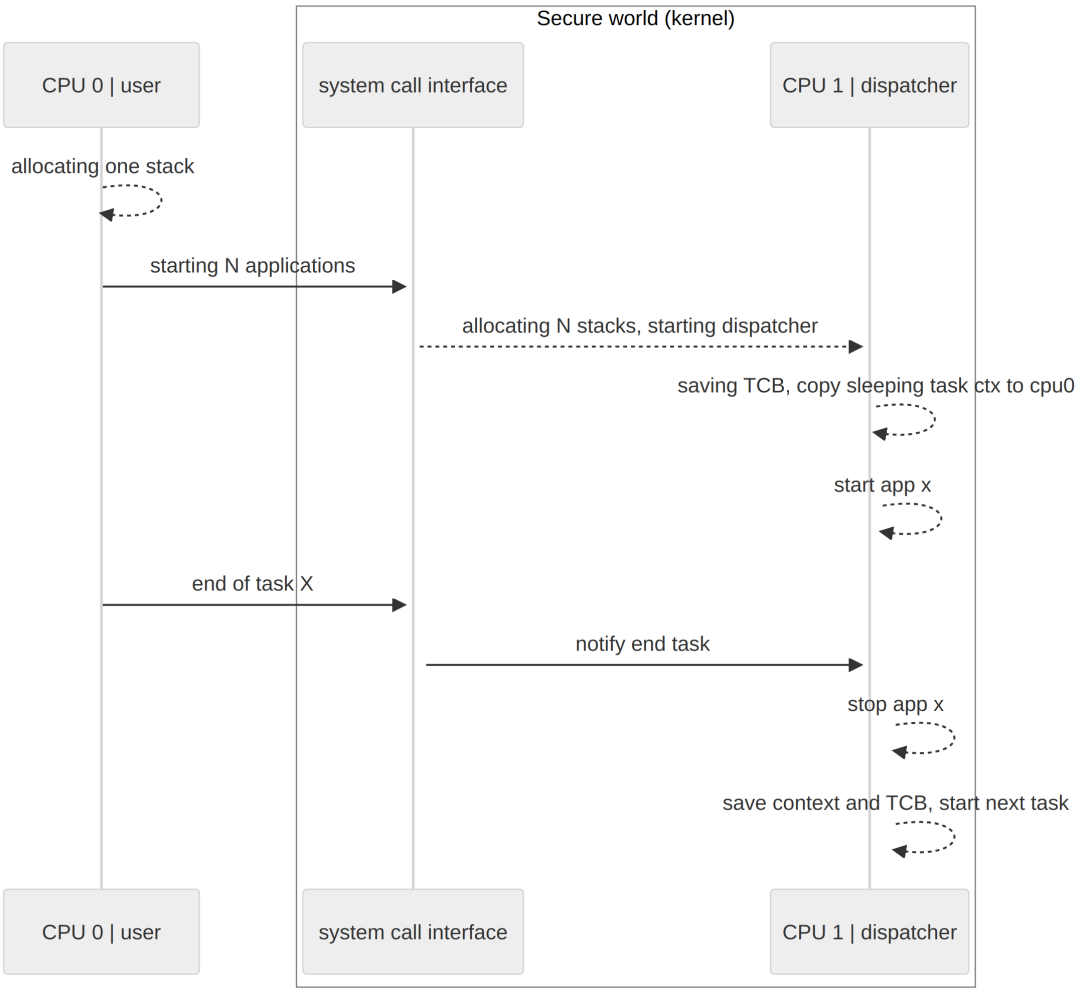

Figure 4.2: Workflow of the preemptive scheduler with yielding capabilities

The code snippet 4.2 is a high level overview of how it is handled in our implementation, for a more complete version, please visit our GitHub[2] repository :

[2] `https://github.com/jbd0101/esp32s3-privilege-separation`

Code Snippet 4.2: User yield high-level code

```
// user function triggering syscall
void user_task(){
    while(;;){
        // task operation
        usr_yield();
    }
}

// kernel function mapped to syscall
void sys_yield() {
    xTaskNotifyGive(dispatchTaskHandle); // notify
       the dispatcher task
}

// kernel world
void sys_task_dispatch(){
    while(;;){
        sys_resume_task(current_task);
        ulTaskNotifyTake(pdTRUE,
           pdMS_TO_TICKS(2000)); // wait for
           notification or 2000 ms
        sys_save_task_context(current_task);
        current_task += 1
    }
}
```

4.1.2 Deep Sleep Capable Scheduler

As detailed in Section 4.1.1, we outline how the scheduler operates seamlessly under continuous user task execution. However, the landscape changes for numerous IoT devices reliant on battery power, necessitating intermittent operation to preserve energy. In such instances, devices enter a deep sleep state when inactive, during which the ESP32 shuts down all components except the Real-Time Clock (RTC) controller and memory. Reactivation occurs solely upon an external trigger on a designated GPIO or prompted by the RTC signal.

The scheduler has been adapted to work with deep sleep, as described hereafter. Upon activation, the kernel configures the allocated number of execution cycles for the tasks before re-entering deep sleep. It also establishes the callback function

that the dispatcher tasks will invoke upon completion of these cycles. Subsequently, the kernel initiates the user task as previously.

The user tasks proceed under the dispatcher's management as per usual. However, once the designated cycles conclude, the dispatcher halts all user tasks and triggers the kernel's callback function. This callback function is responsible for computing the duration of the user tasks' activity, which is essential for analytics or billing purposes. Additionally, it ensures the preservation of data in flash storage when necessary before commencing a new sleep cycle.

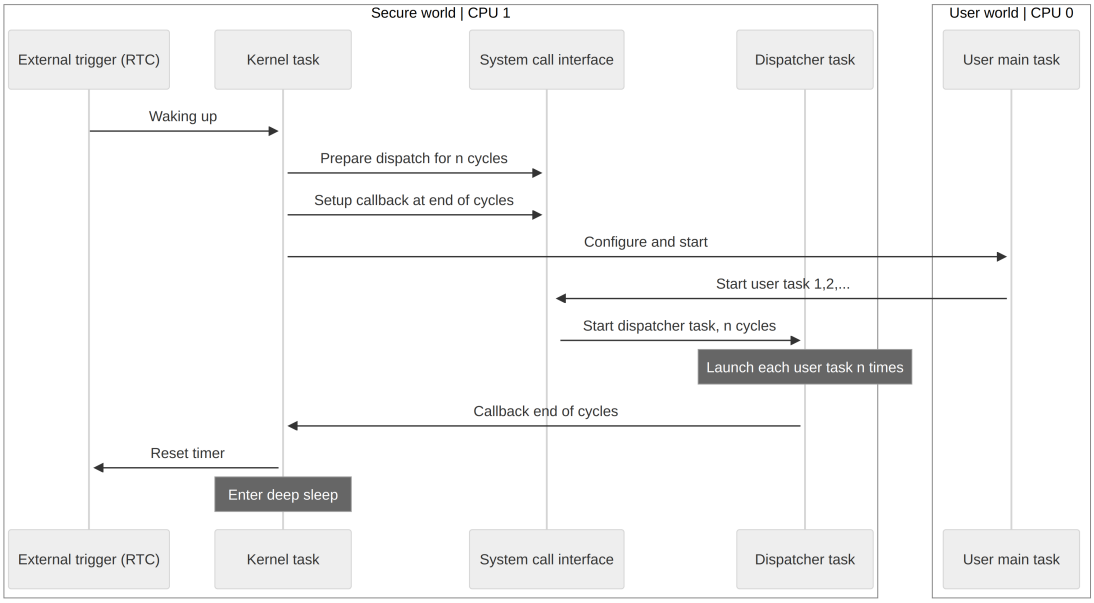

Figure 4.3: Adapted scheduler for deep sleep

4.1.3 Error Handling

The PerMiSsion control (PMS)[3] allows for capturing and triggering an interrupt in the kernel space if user code attempts to access an unauthorized memory zone. Additionally, our timer mechanism prevents user tasks to trigger a watchdog interrupt because it times out before the watchdog gets triggered. However, user code can still generate an error that results in a complete reboot of the ESP system. It should be noted that fatal errors on an ESP32 cannot be intercepted with an

[3]This abbreviation is unconventional, yet it is considered the "official" one, as indicated by this forum post: `https://esp32.com/viewtopic.php?t=40040`

interrupt handler [4].

To address this limitation, we introduce two new elements: a simple current task pointer that is preserved after a reboot, and a task mask that is stored in the NVS. This tasks mask is an array of bits initialized to 1, and when a task gets blacklisted, the bit corresponding to its index is turned into a 0. Before the task dispatcher starts, the kernel checks if the saved task pointer corresponds to any of the tasks. If the task pointer matches one of the tasks, it indicates that the previous shutdown was not executed by the kernel and was likely caused by the associated user task. Consequently, the task dispatcher blocks this user task by blacklisting it in a task mask stored in the NVS.

Our testing demonstrates that this method is highly effective in identifying and banning problematic user tasks. However, we encounter some false positives when the ESP is unplugged from the power source; upon reboot, the last user task is incorrectly identified and banned.

One advantage of this approach is its very low overhead: the pointer is updated during each user task switch, and there is minimal overhead due to two reads from the NVS during boot.

[4] `https://docs.espressif.com/projects/esp-idf/en/v4.3.1/esp32c3/api-guides/fatal-errors.html`

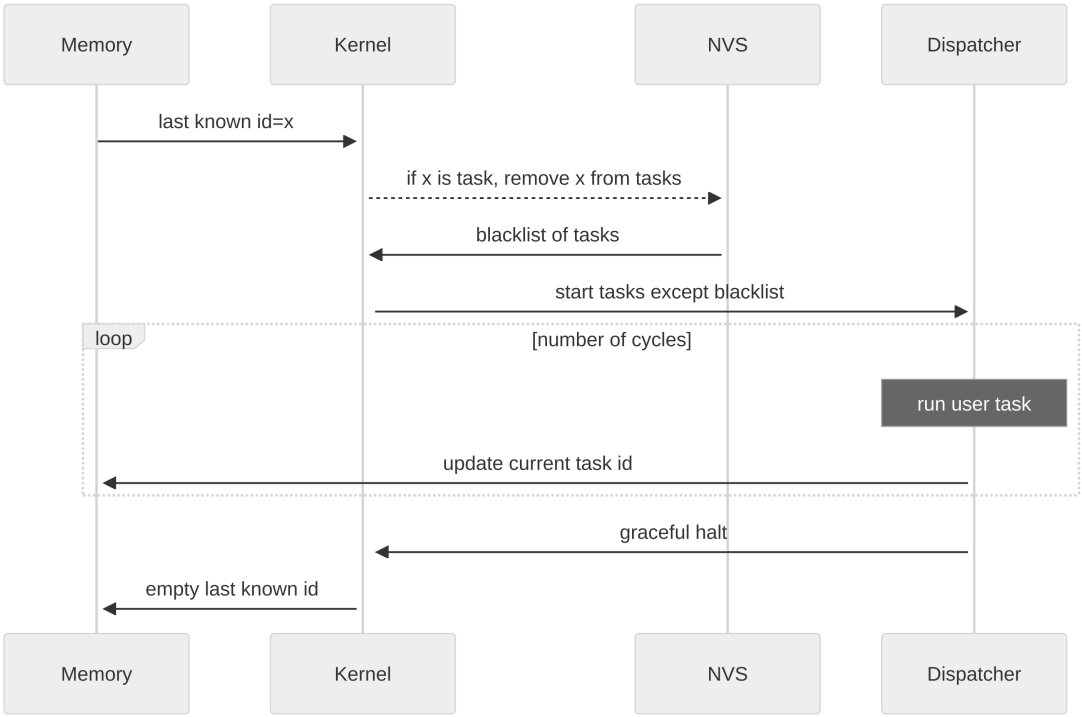

Figure 4.4: Diagram of the automatic user task banning

4.2 Memory Usage

Concerning the RAM usage, the ESP32S3 features 512 KB of Static RAM (SRAM) and up to 8 MB of Dynamic RAM (DRAM), with our specific configuration incorporating 2 MB of DRAM. The user-accessible Instructions RAM (IRAM) memory is fixed at 2048 bytes, complemented by 1 MB of DRAM. These memory segments are exclusively accessible to user space, and they are allocated for all user applications. Given that each user application has full access to this memory, it is essential to maintain low memory usage to allow the kernel to manage the stacks of multiple applications concurrently.

A key limitation to the number of possible applications is that each user application allocates DRAM from the kernel, which can constrain the overall system performance and scalability. This issue can be mitigated by utilizing external Pseudo-Static RAM (PSRAM) [5], which can store the stack data of idle user applications, thereby reducing pressure on the internal DRAM and enhancing the

[5] `https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-guides/external-ram.html`

system's capability to handle more applications efficiently. For further details, refer to the ESP IDF documentation on external PSRAM.

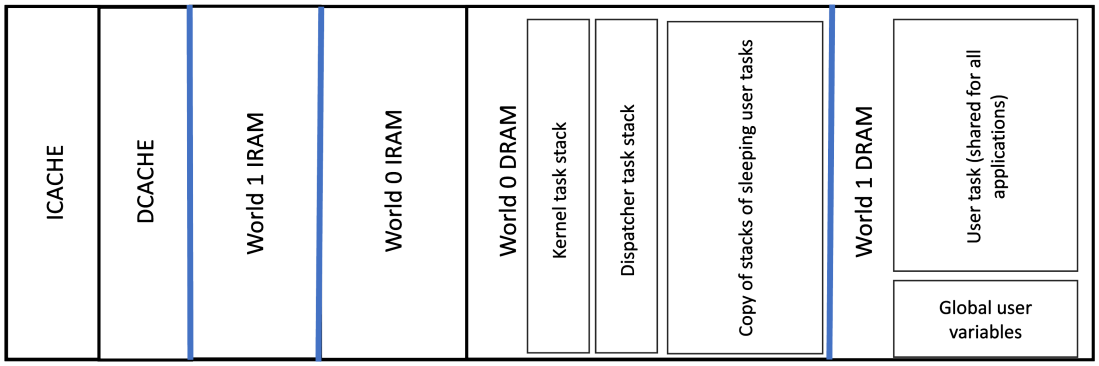

Figure 4.5: User and kernel memory separation SRAM and DRAM — not on scale

Figure 4.5 depicts various memory regions. The Instruction Cache (ICACHE), Data Cache (DCACHE), and IRAM are stored in SRAM. Additionally, the user stacks can be transferred to external PSRAM. The blue lines indicate the paddings ($\approx$ 256 Bytes) required for the PMS.

The available flash memory is organized into several partitions, each serving distinct purposes:

- `NVS` Partition : This Non Volatile Storage (NVS) partition is utilized by the ESP as default NVS storage. It is particularly useful for certain drivers, such as the Wi-Fi driver. Wi-Fi configurations are stored in this partition, allowing the Wi-Fi driver to access previously stored configurations using the `esp_wifi_get_xxx` APIs upon power-up or reboot. This negates the need for reconfiguration from scratch. While the NVS partition is not mandatory and can be removed to free up space, doing so would lead to performance degradation in several modules.
- `phy_init` Partition: This optional partition stores calibration data for the Radio Frequency (RF) module. It allows the device to save data from previous calibrations and perform partial calibrations, which are quicker than full calibrations, though slightly less precise. Similar to the NVS partition, this can be omitted to conserve space, albeit at the cost of reduced performance.
- `factory` Partition: This partition holds the kernel firmware along with the system call interface. In its current implementation, it occupies 1 MB, but this size can be adjusted based on the growth or optimization of the application.

- `user_app` Partition: This partition contains the binary for client applications and the user dispatching task. It must be adequately sized to accommodate all user code.
- `user_app_sec` Partition: This partition mirrors the size of the `user_app` partition and serves as the passive storage for the updated binary during OTA updates.
- `ota_data` Partition: This partition stores the address of the current partition used for the user application, facilitating the OTA update process.
- `static_data` Partition: This partition is designated for storing client secrets.
- `NVS Key` Partition : This partition stores the keys required when NVS encryption is enabled. Its usage is described in section 4.3 of this chapter.

It is crucial to note that partition sizes cannot be altered via OTA updates. Therefore, partition sizes must be carefully chosen during the initial setup. The recommended OTA process by ESP IDF demands significant memory due to the requirement for an unused partition. This memory usage issue can be mitigated by implementing a two-step OTA process: first, changing the `ota_data` partition to boot from a smaller partition, which will then download and flash the new firmware onto the main partition.

The ESP32S3 is available in various versions, with flash memory ranging from 4 MB to 32 MB. Manufacturers can adjust the number of user applications and disable the OTA feature to operate within the 4 MB constraint. This flexibility allows for optimization based on specific application requirements and available memory.

4.3 Secrets Handling

Given the frequent absence of physical oversight for the device, ensuring users can securely store sensitive data like private keys, passwords, and Personally Identifiable Information (PII) became paramount. To address this requirement, we employ the NVS capabilities of the ESP. However, it must be noted that this library is not exclusive to the privilege separation framework; it is also accessible within the ESP IDF framework.

The NVS library facilitates the storage of key-value pairs within the device flash memory through the allocation of a NVS partition in the partitions file. Its primary function is to accommodate numerous small pairs rather than a few lengthy values. The permissible value types include integers (ranging from 8 to 64 bytes, signed

or unsigned), null-terminated character strings, and binary blobs. String values can extend up to 4000 bytes in length, while blobs can reach a maximum of 508 KB [6]. Key strings are limited to a maximum of 15 characters. As NVS operates as a key-value store, duplicate keys are disallowed. To address this constraint, ESP introduced namespaces of up to 15 characters, enabling multiple uses of the same key within separate namespaces. For instance, both User 1 and User 2 can employ the identical key "rsa_private_key", which will be distinguished by their respective namespaces, "user1:rsa_private_key" and "user2:rsa_private_key".

While the NVS partition supports encryption, it does not seamlessly integrate with the default ESP flash encryption system. Data within NVS partitions can undergo AES-XTS encryption, adhering to the IEEE p1619 standard [14], where each entry is treated as a single sector, and its relative address is utilized as the sector number in the encryption algorithm. The encryption key for the NVS partition is stored in a separate partition, safeguarded by the ESP flash encryption system. Activating this encryption mechanism is hence required to enable NVS encryption, which is not done by default.

When users wish to register new tasks, we allocate to them a randomly generated namespace, and they furnish their secrets for storage on the flash. Accessing these secrets necessitates users to invoke a custom syscall function, which triggers an interrupt to the kernel world. Subsequently, the kernel retrieves the requisite secret from the NVS and returns it to the task. At initialization, we map each task with a namespace in order to avoid names overlap and to prevent users to access other another user's secret. Dynamic addition, removal, or updating of secrets during program execution is currently not supported.

It is advisable to minimize the number of requests made to flash memory. When feasible, the client's application logic should retrieve the necessary secrets in a single request and then store these secrets in the stack. This approach ensures that the secrets are preserved during task switches and are protected from access by other users.

To further substantiate our claim, we conducted a series of benchmarks to measure the performance of accessing a total of 16 KB of memory from the flash partition responsible for handling secrets. This experiment involved varying the number of secrets accessed, ranging from 32 to 512. Figure 4.6 depicts a clear linear decrease in performance correlated with the increase in the number of accesses. Specifically, each time the number of accesses was doubled, the time required to retrieve all the memory was also doubled, illustrating a direct proportionality between access

[6] `https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/storage/nvs_flash.html#keys-and-values`

frequency and time consumption.

The benchmarking process was carried out using the API provided by the Espressif IDF [7]. This API facilitates the setting and retrieval of blobs, which is the recommended method for accessing complex data structures in this context.

However, it is worth noting that while the use of blobs is advised for its general applicability and ease of use, there are scenarios where employing more specialized APIs can yield superior performance. For instance, directly using APIs tailored to specific data types, such as signed or unsigned integers and strings, can offer enhanced efficiency. These specialized APIs bypass some overhead associated with more generic blob operations, leading to faster access times in cases where the data type is known and fixed.

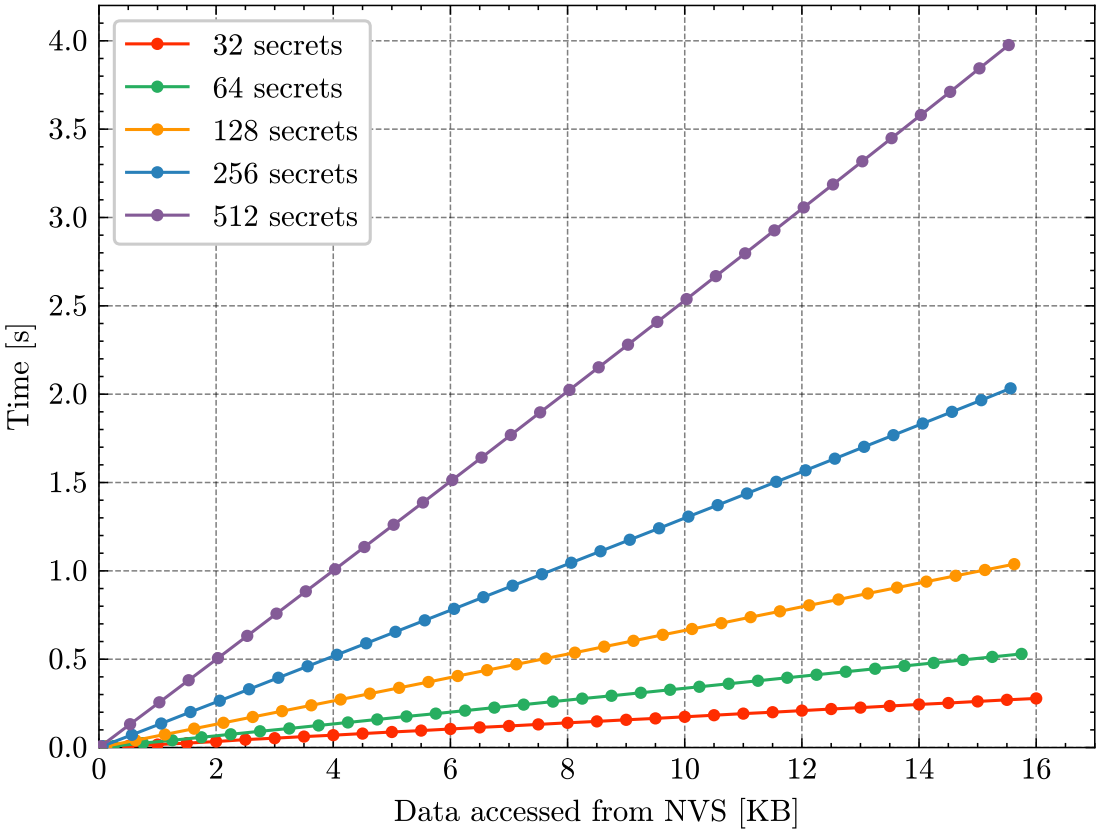

Figure 4.6: Benchmark of the time required to access 16 KB of data with varying number of accesses

[7] https://docs.espressif.com/projects/esp-idf/en/v4.4.3/esp32/api-reference/storage/nvs_flash.html

4.4 Pipeline

The user code needs to be able to access the sensors and the actuators to accomplish this task. We could not give direct hardware access for multiple reasons :

- As the PMS is only about registers, there would not be possible to give permissions on only a subset of the GPIOs.
- It is unpractical to change the PMS settings between each user task switch (as one user would not necessarily have access to the same GPIOs)
- Each user task reading the same sensor would be inefficient: the user code will not know whether the sensor needs to be put in sleep mode or not

To address this issue, sensor readings can be concealed behind system calls, as demonstrated in our code. This system mitigates the previously mentioned challenges, but is not entirely effective for the following reasons:

1. It is difficult to put the reading in the cache.
2. It is not flexible, as each sensor needs a new system call endpoint.
3. It negatively impacts the code quality: it is a bad development practice to see the reading of the sensor in kernel code. The kernel should be as universal as possible.

Therefore, we developed a flexible solution based on queues: the kernel creates a queue for each user task. Then, each user task subscribes to events (for example: `TEMP_SENSOR`). When the kernel creates a new data, it is automatically added to each subscribed user pipeline. The user code can consume an event whenever it wants. The advantage of this approach is that one API works for any kind of data (as long as it is not bigger than the packet size). The non-secure world is in charge to read the data periodically (or on trigger) for all the user codes. A user code that would not consume the data would not hurt the system, old data being overwritten in the queue when overflow is happening. The pipeline is unidirectional, this design rationale comes from the observation that data transmission from kernel to user is a more common scenario, given the kernel's typical access to sensor data, thus reducing the necessity for bidirectional communication. Although the integration of bidirectional communication is technically feasible, it was purposefully omitted to mitigate associated overhead costs. Figure 4.7 provides an high level view of the pipeline.

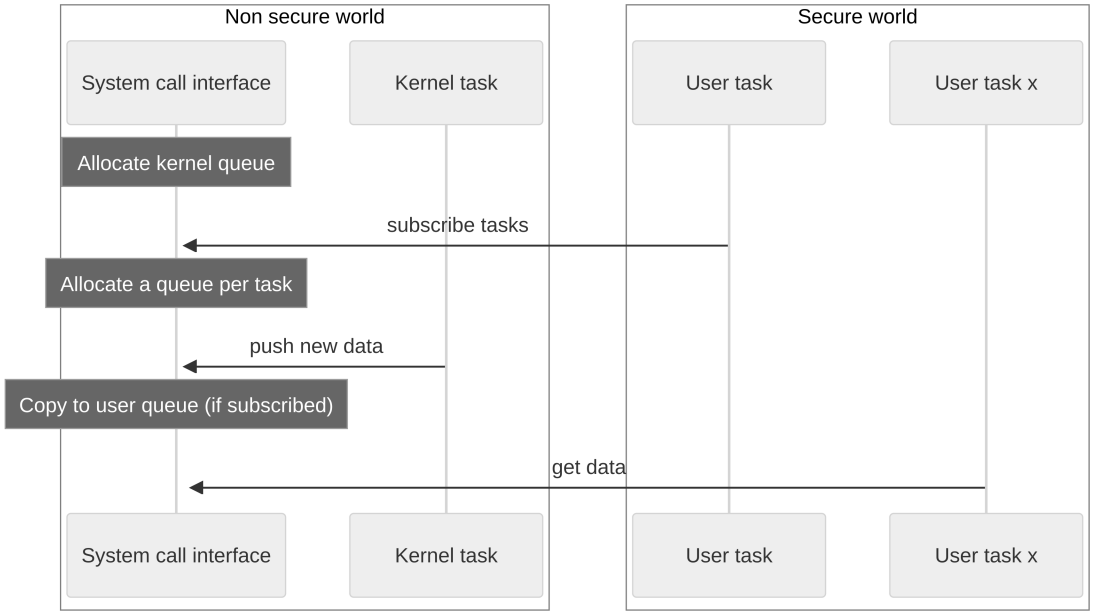

Figure 4.7: High level view of unidirectional pipeline communication

The primary objective of the pipeline is to establish a shared memory interface between the kernel and user domains, addressing the current limitation where data exchange mandates task instantiation and passing arguments, incurring substantial overhead—especially evident when transmitting numerous small data objects. In our system, the pipeline is accessible to all user tasks, rendering it unsuitable for transmitting sensitive or confidential information to a specif task.

4.5 Practical Deployment and Economic Model

In this section, we explore potential commercial use cases and demonstrate how our solution could function in real-world scenarios. This discussion aims to provide readers with a foundational understanding of how our implementation could be monetized, along with the associated requirements and limitations of the solution.

Firstly, we outline the prerequisites for clients who wish to upload their code to the device, detailing the steps and options available to them. This includes an examination of the tools, protocols, and procedures necessary for successful code deployment. Additionally, we discuss the responsibilities of the manufacturer, emphasizing the importance of providing comprehensive APIs, thorough documentation, and a robust development environment. These elements are critical to

ensuring that clients can efficiently develop, test, and deploy their applications on the device.

Subsequently, we propose a set of metrics for billing clients based on the device's constrained resources. These metrics take into account factors such as memory usage and processing time, offering a fair and transparent billing structure. By considering these resource constraints, we aim to create a billing model that reflects the actual usage and demands placed on the device, thereby incentivizing efficient and optimized application development.

Through this detailed examination, we aim to illustrate the practical implications of our solution, providing a clear roadmap for both clients and manufacturers. This helps to ensure that our implementation is not only technically sound but also commercially viable.

4.5.1 Deploying Client's Code

As in cloud operations or any scenario where the client does not control the entire infrastructure, establishing a strong mutual trust between both parties is paramount. Clients must trust the manufacturer to safeguard their proprietary code and deployment secrets, ensuring that these remain private and unaltered post-deployment. Concurrently, manufacturers need to trust that clients will not attempt to disrupt the device or uncover other clients' secrets. This mutual trust is bolstered by the various protection mechanisms embedded within the framework, which aim to mitigate such risks.

The development environment is minimal yet functional. For effective development, clients would benefit from access to the complete codebase, including the latest updates, kernel task management functions, and the default system calls API in an open sourced way. Clients can then integrate their code within this project, ultimately submitting the task code and any custom system calls to the manufacturer. However, without physical access to the target device, thorough testing can be challenging, as the code interacts with specific sensor configurations. This limitation impacts the user experience and is discussed in detail in the limitations section (6.1).

Once clients have developed their applications, they must submit their code and secrets to the manufacturer. The manufacturer integrates this with the existing codebase, along with other clients' code, and triggers an OTA update to remotely deploy it to the device. Clients are responsible for monitoring and reporting the required stack size for their applications to ensure sufficient memory allocation during task creation. This process underscores the need for meticulous resource management and coordination between clients and manufacturers.

Providing comprehensive and up-to-date documentation is crucial for manufacturers. This documentation should guide users on how to query various sensors and detail the available system calls. Clear, detailed instructions enable users to effectively interact with the device's functionalities, ensuring a smoother development and deployment process.

Regarding sensor data transmission, multiple strategies can be employed. One approach involves the kernel periodically reading sensor data and pushing it to a queue, where subscribed user tasks can access it. This method is particularly advantageous for sensors that require significant power to read data. Alternatively, system calls can be created to allow users to poll data as needed, though this can be resource-intensive. A third approach is to grant user-level privileges for accessing specific modules, but this is not feasible for all use cases. It currently lacks the capability to grant permissions selectively to individual user tasks, potentially leading to unauthorized data access or module behavior modifications.

These considerations highlight the complexity and importance of carefully designed interaction protocols and security measures to maintain the integrity and functionality of the system while enabling effective client applications.

4.5.2 Billing

The nature of embedded devices is to be inherently constrained in resources, whether in terms of time, power, or storage space. Given these limitations, it is crucial to explore various methods for billing users based on their device usage. Our study delves into several potential billing strategies.

Among the various resources, time stands out as the most critical, as it directly influences the functionality of an application. A fair method for measuring a client's code runtime is to track the number of ticks that elapse between the moments when the task context is reloaded and when the task notification or watchdog timer is triggered. Fortunately, the FreeRTOS API provides a useful function for this purpose, `xTaskGetTickCount` [8]. During system initialization, we could allocate an array of `uint64_t` counters, which would increment each time a task runs by measuring the tick count before and after the task's execution, and then subtracting these values. The dispatcher task would then update the total tick count in the counter. This data could be sent back to the owner's server on a daily, weekly, or monthly basis, depending on the client's requirements, to provide an up-to-date view of their resource consumption. This approach enables the device owner to monetize tick usage according to their commercial strategies and device costs.

[8] `https://www.freertos.org/a00021.html#xTaskGetTickCount`

Storage constraints manifest in three main forms: the size of the client's code, the stack required by the task, and the size of the secrets that need to be stored. Addressing the stack size requirement is relatively straightforward; the user simply needs to specify their stack requirements to the device owner during task initialization, ensuring that these resources are appropriately allocated. However, determining the size of the client's code is more complex. A basic method would involve billing based on the C code size, charging for each byte used. This approach is flawed because it does not account for comments in the code, which do not occupy space in the binary, leading to potential overcharging. Additionally, it fails to consider compiler optimizations, which can reduce the actual size of the code in the binary, meaning users might be charged for non-existent or optimized-out code.

Our proposed solution, though primitive, involves creating a "skeleton" application that contains the basic code for task dispatching and application maintenance by the device owners. By building this skeleton application, we can determine its packed size. To know the size of a new user task, we add it to the skeleton application, build the combined application, and then subtract the size of the skeleton application from the total size of the binary. This method provides a more accurate measure of the user's task size.

Lastly, given the limited flash memory that must be shared among all users for storing secrets and task code, it is essential to charge for the secrets per byte used. Users should be reminded that it is more efficient to store their secrets in one large contiguous block of bytes rather than reading from multiple addresses, as flash storage access times are significantly slower.

By implementing these strategies, we can effectively manage the constrained resources of embedded devices while providing a fair and transparent billing system for users based on their actual usage. This approach ensures that device owners can optimize resource allocation and monetize their devices appropriately, while users benefit from a clear and fair billing structure.

4.5.3 Alternative Pricing Approaches

Currently, our scheduler implementation exhibits a rigid structure, treating all user tasks with equal priority. This uniform approach does not accommodate varying levels of task urgency or user requirements. To address this limitation, we propose implementing a priority-based scheduling system. Such a system would allow users to assign different priority levels to their tasks, potentially by paying a premium to secure more frequent CPU access for high-priority tasks. This model introduces a more dynamic and responsive scheduling mechanism, aligning resource allocation with user needs and willingness to pay.

Furthermore, we could draw inspiration from the pricing models used by cloud service providers such as Amazon Web Services (AWS). Specifically, AWS offers Spot Instances, which allow users to bid on unused EC2 capacity at significantly reduced rates compared to standard on-demand instances.

Although implementing a bidding system for CPU time on our device may not be feasible, we can adopt a similar concept where users pay based on the priority level required by their applications. This approach ensures that our scheduler can offer different tiers of service, thus enhancing flexibility and catering to a broader range of user scenarios. Users with high-priority tasks would be able to pay more to ensure timely execution, while cost-sensitive users could opt for lower-priority scheduling at a reduced cost.

By integrating a priority-based scheduler and incorporating tiered pricing strategies, our system can evolve to provide a more tailored and efficient resource management solution. These enhancements will be explored in greater detail in section 6.2, where we will delve into the potential implementations and benefits of such a system. This future work aims to transform our scheduler into a more versatile tool, capable of adapting to diverse user requirements and optimizing overall system performance.

Chapter 5

Evaluation

In this chapter, our focus shifts towards a thorough evaluation of our implementation in comparison with both the ESP IDF and the privilege separation framework. Through a structured series of experiments, we delve into measurements of power consumption and execution times to discern differences among the solutions. Additionally, we carefully analyze our findings to gain meaningful insights into the performance and effectiveness of each approach. By offering a detailed examination of our experimental results, we aim to provide a nuanced understanding of the strengths and limitations in the evaluated frameworks.

In our investigation, we opted to focus on two key aspects: electrical current consumption and execution times. These factors are closely linked with the inherent characteristics of embedded devices. Embedded systems, characterized by real-time requirements and the necessity to conserve power, face the challenge of operating reliably for prolonged periods without a consistent power source.

Understanding the relationship between these metrics is crucial in the context of embedded systems. Current consumption serves as a vital indicator of a device's energy efficiency, directly impacting its ability to function over extended durations, especially in scenarios with limited power availability. By scrutinizing and optimizing current consumption, developers can enhance the longevity of embedded devices and reduce their dependence on external power sources.

Similarly, execution times play a significant role in assessing system performance, particularly concerning real-time responsiveness. Efficient utilization of computational resources is essential for meeting strict timing requirements and ensuring prompt responses to external stimuli. Minimizing execution times not only improves system efficiency but also enhances user experience and facilitates seamless integration of embedded devices across various applications.

5.1 Methodology

The experimental process begins with an assessment of resource consumption in "blank" applications to establish a baseline for subsequent evaluations. Following this, we measure the impact of the two frameworks and specifically system calls on CPU-intensive computations with high throughput requirements. Afterward, we evaluate the effectiveness of our multitasking implementation and its preemptive scheduling optimization by comparing it to a native IoT Development Framework (IDF) application which performs the same computations, without the security enhancing provided by the privilege separation framework. Finally, we benchmark two "real-world" applications that emulate typical usage scenarios of embedded systems to assess the practical performance of the Privilege Separation Framework (PSF).

The electrical current measurements were done on an ESP32-S3-DevKitC-1 development board, with an ESP32-S3-WROOM-1[1] by supplying a 5.0V input and measuring the current with a Keithley 2450 source meter [2]. This scientific instrument is designed for precise electrical measurements and sourcing. It is capable of measuring and sourcing voltage, current, and resistance with high accuracy. It allowed us to measure the current consumption, with a sampling rate of more than 100 times per second and a precision of more than 6 digits. Due to our usage of a development board, the consumption includes the consumption of the ESP32 as well as all the other components of the development board (voltage regulators, passive components, ...). Figure 5.1 depicts our experimental setup.

[1] `https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/hw-reference/esp32s3/user-guide-devkitc-1.html`

[2] `https://www.tek.com/en/datasheet/smu-2400-graphical-sourcemeter/model-2450-touchscreen-source-measure-unit-smu-instrument`

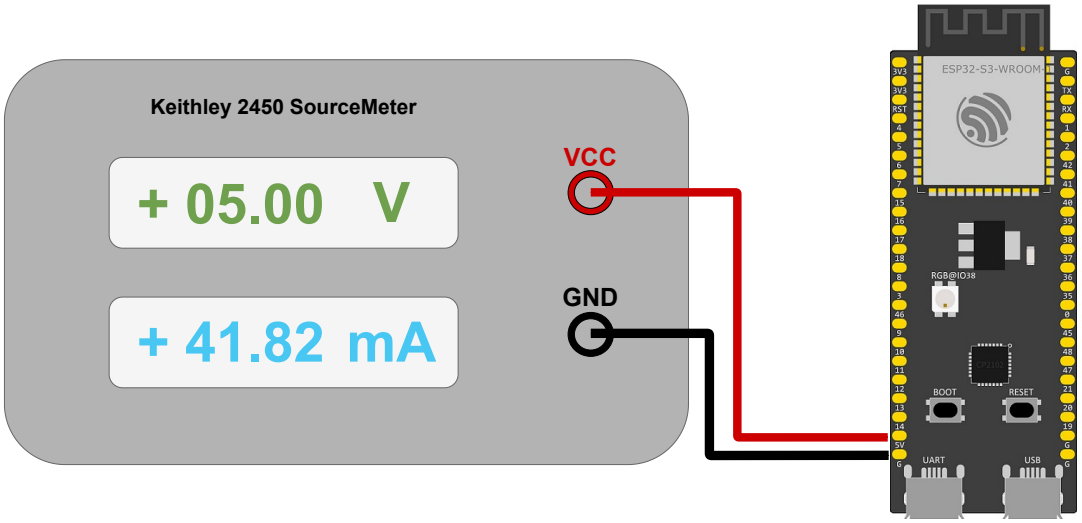

Figure 5.1: Experimental setup for energy and current consumption measurements[3]

5.2 Baseline Consumptions

To examine how much current the device consumes when it is idle—a usage that cannot be avoided without implementing optimizations like deep sleep or adjusting the clock—we ran tests using both the IDF v4.4 and PSF v4.4. In this setup, we developed an application that solely initializes the device without launching any tasks or activating any peripheral functions or modules. Specifically, for the IDF application, it consists of a main loop that repeatedly executes without performing any operations. Similarly, the PSF application mirrors this behavior; it initializes and executes a main loop within the kernel environment, without initiating any tasks in the user space. This approach enables us to compare power consumption between the two frameworks in a straightforward manner, shedding light on their efficiency during idle periods.

[3]Credit for the development board diagram : `https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/_images/ESP32-S3_DevKitC-1_pinlayout_v1.1.jpg`

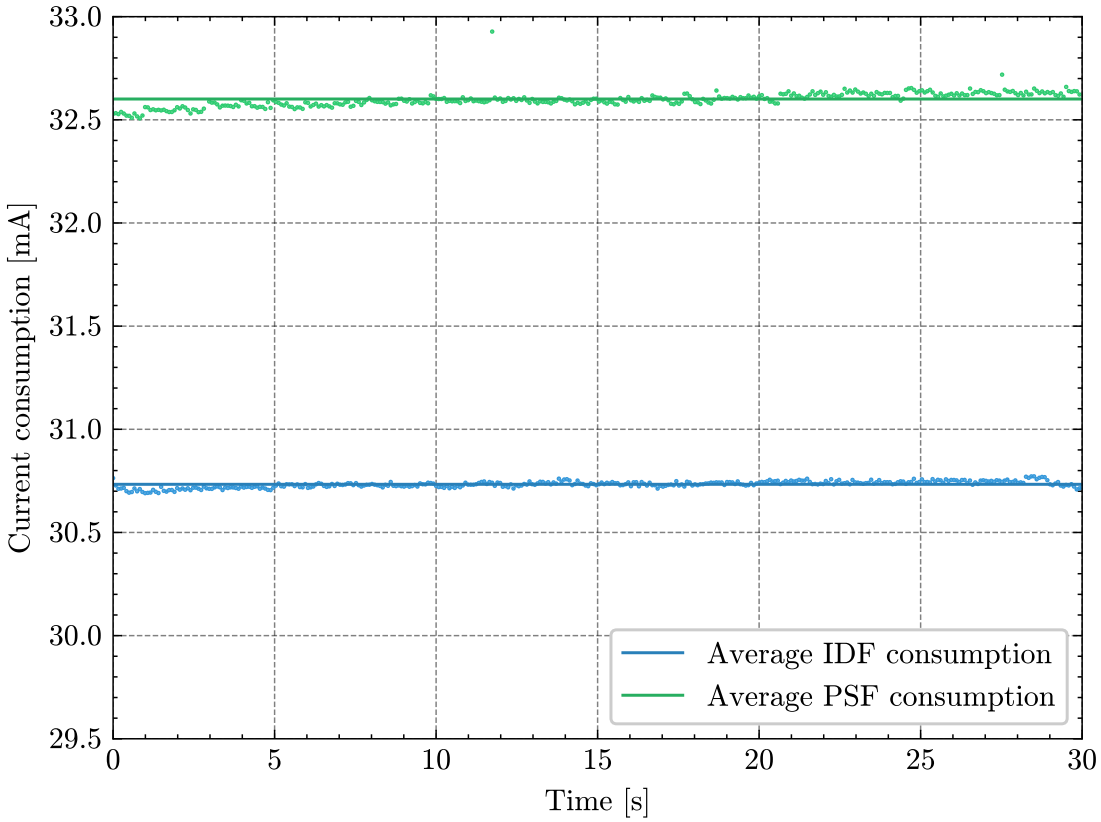

Figure 5.2: Comparison of baseline consumptions between the IDF and the Privilege Separation Framework

One can clearly see on figure 5.2 that the baseline consumption of the PSF is approximately 1.9 mA higher than the baseline consumption of the IDF, which an average current consumption of 30.7 mA for the IDF and an average current consumption of 32.6 mA for the PSF. When comparing these two environments, we observe the following distinctions: the activation of permission control and the world controller, and the initiation of core 1 to execute kernel code (whereas in the blank IDF, core 1 remains unused while core 0 waits for user tasks). We conclude that these two factors contribute to the 6.19% increase in current consumption.

5.3 CPU Intensive Computation

To investigate the influence of the various frameworks on device performance during computational tasks, we conduct an experiment to quantify the current consumption while executing addition operations involving millions of integers.

Although the precise count of integers varies across scenarios, the fundamental approach remains consistent. A `for` loop iteratively calculates the sum of integers, as described in Code Snippet 5.1.

Code Snippet 5.1: Sum function

```
uint32_t compute_n_first_sum(int n){
    uint32_t sum = 0
    for (int i = 0; i < n ; ++i){
        sum += i;
    }
    return sum;
}
```

In order to thoroughly benchmark each possibility and fully understand the performance differences, we divide this experiment into four distinct scenarios. These scenarios are as follows:

1. An application using IDF where the main function loops and performs 40 million additions, waits for 1000 ms, and then repeats.
2. An application using PSF where the kernel does not launch any user tasks but initiates a kernel task that performs 40 million additions, waits for 1000 ms, and then repeats.
3. An application using PSF where the kernel launches a user task that performs 40 million additions, waits for 1000 ms, and then repeats.
4. An application using PSF where the kernel launches a user task that performs **1 million** additions, waits for 1000 ms, and then repeats. In this scenario, the user task makes a system call to spawn a task in the kernel space to execute the additions, and then returns the result to the user task. This behavior is designed to simulate a user task that frequently needs to execute code protected by the kernel.

For these measurements, we record the current consumption over time, as illustrated in Figures 5.4, 5.5, 5.6, and 5.7. Additionally, we derive data on the time required by each scenario to complete a full set of additions, presented in Table 5.1. Understanding the energy consumption of each scenario is crucial for assessing their feasibility in real world battery-powered devices, to this end, we compute the energy consumption for each scenario over nine iterations and reported it in Table 5.2.

Our methodology for this calculation is as follows: we evaluate the cumulative

energy consumption from the onset of the initial current spike to the onset of the tenth spike (indicated by vertical black lines on the charts). We then compute the integral of the current using the composite trapezoidal rule and multiply it by the constant voltage (5V) used in the experiments. This approach provides us with the total energy (in watt-seconds or joules, denoted Ws) required to compute 9 sums. The trapezoidal rule approximates the area under the graph of a function as a series of trapezoids. [4]. For the computation, we use the uniform grid variant of the Trapezoidal rule. The approximation of the power consumption is described in Equation 5.3.

$$
\begin{aligned}
E &= \int_{t_1}^{t_2} P(t)\,dt \\
&= \int_{t_1}^{t_2} V \times I(t) \\
&= \int_{t_1}^{t_2} 5 \times I(t) \\
&\approx 5 \times \frac{\Delta t}{2} \sum_{i=1}^{n} (I(t_{i-1}) + I(t_i))
\end{aligned}
$$

Equation 5.3: Approximation of energy using the trapezoidal method

In each plot, we represent the measurement samples using red scatter plots and blue step plots. Unlike a standard line plot that connects data points with straight lines, a step plot creates a series of horizontal and vertical lines, producing a staircase-like appearance. This method is particularly useful for visualizing data that changes at discrete intervals, such as time series data. Additionally, we include a green dotted line to represent the mean current consumption. The black vertical lines denote the start and end points of our cumulative energy consumption measurement.

[4] `https://en.wikipedia.org/wiki/Trapezoidal_rule`

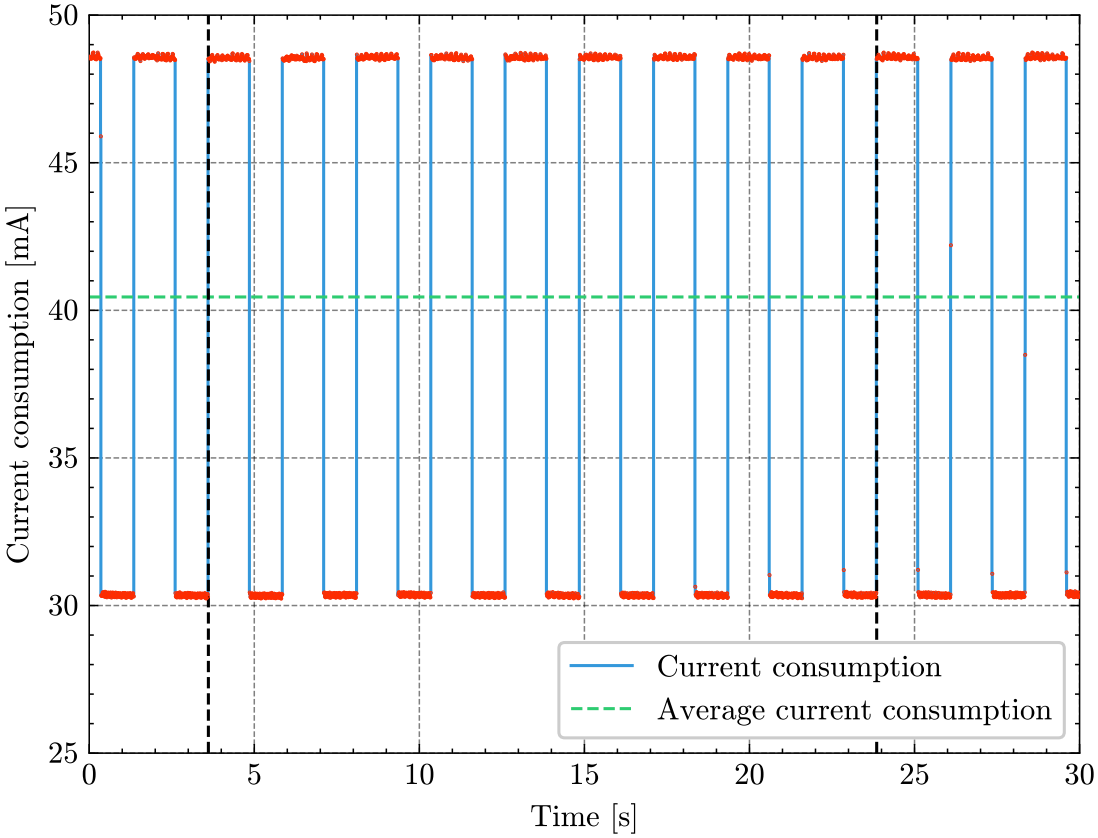

Figure 5.4: Sum of **40 million** integers with 1s delay between each loop on IDF. This task needs 4.096 Ws of energy and 20.24 seconds to accomplish the 9 additions.

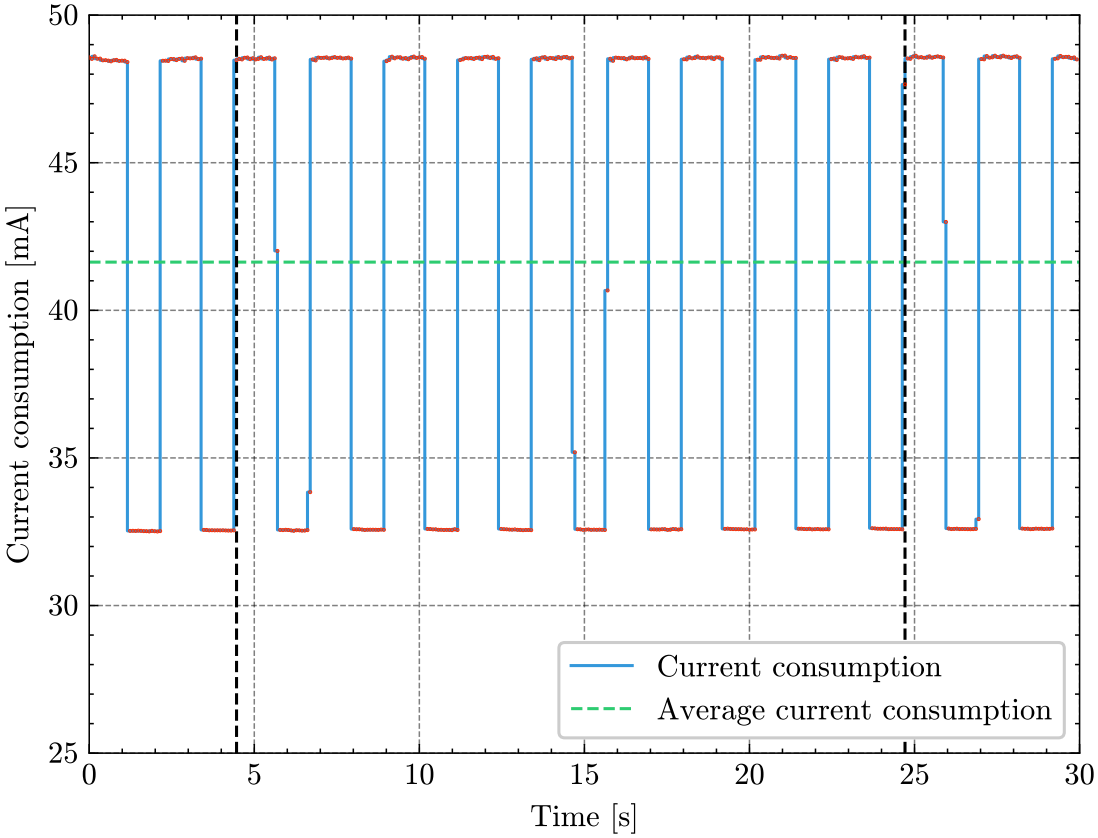

Figure 5.5: Sum of **40 million** integers with 1s delay between each loop on PSFthe computation runs on the kernel task.
This task needs 4.180Ws of energy and 20.24 seconds to accomplish the 9 additions.

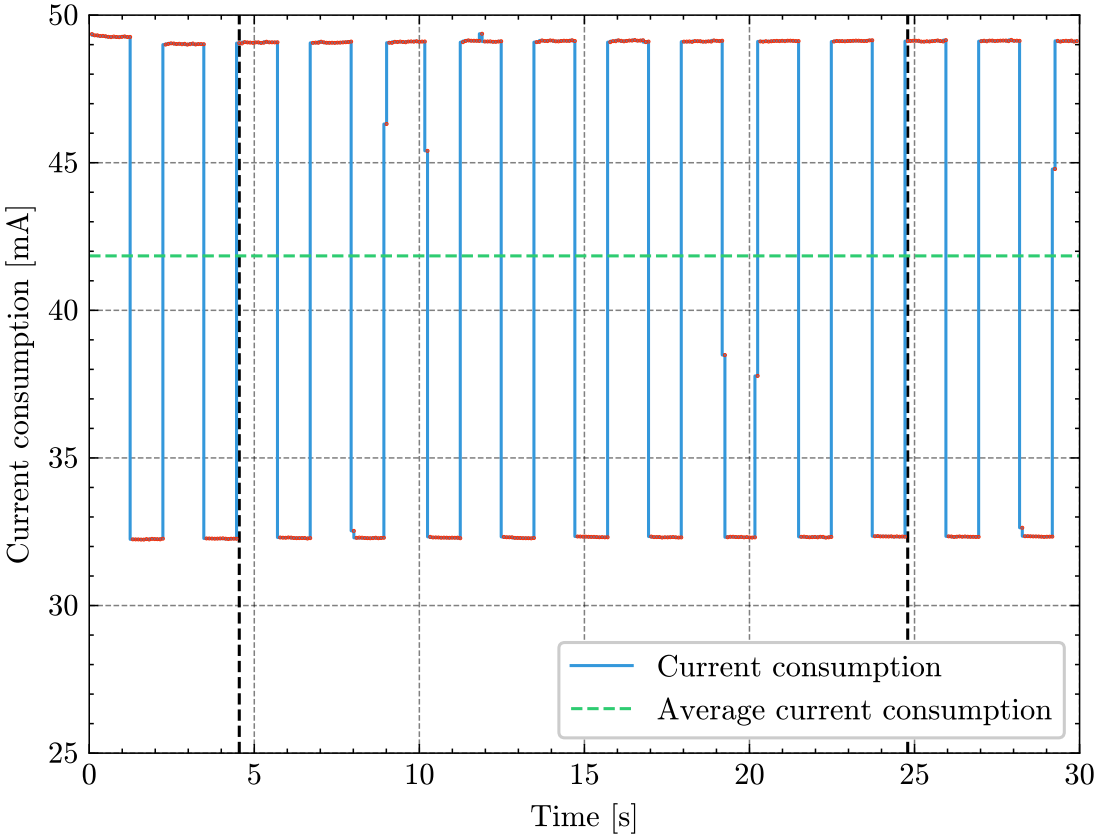

Figure 5.6: Sum of **40 million** integers with 1s delay between each loop on PSF the computation runs on the user task.
This task needs 4.2 Ws of energy and 20.24 seconds to accomplish the 9 additions.

Comparing the IDF (Figure 5.4) with kernel addition (Figure 5.5) shows a consistent difference in current consumption averages, similar to that observed in the baseline scenario (Figure 5.2). Kernel addition registers approximately 1.9 mA higher current consumption. Secondly, when comparing kernel addition (Figure 5.5) with user addition (Figure 5.6), the averages are nearly identical, and peak and idle periods match. This is as expected, since computational tasks in either context should have minimal impact when using native C code without triggering syscalls.

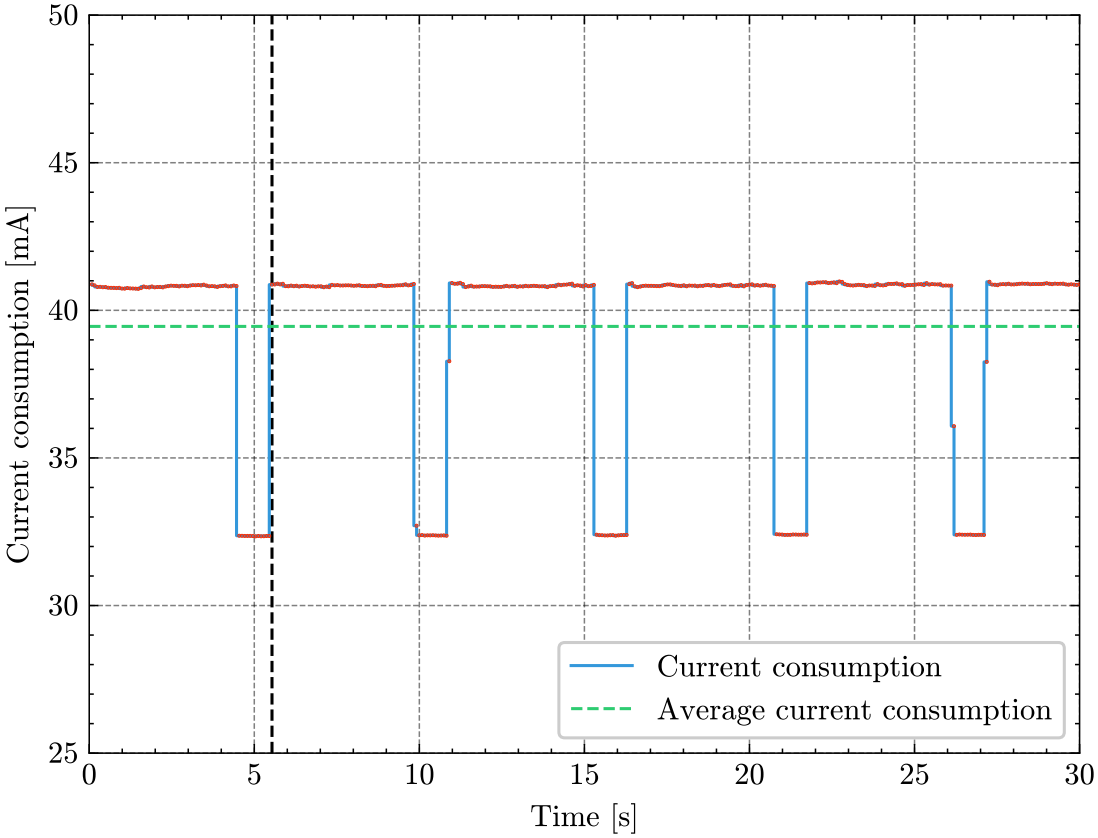

Figure 5.7: Sum of **1 million** integers with 1s delay between each loop on PSF the computation triggers syscalls from the user to the kernel task.
This task needs 9.57 Ws of energy and 48.75 seconds to accomplish the 9 additions

Particularly interesting is the comparison between user addition and syscall addition (Figure 5.7). In the latter, a user task executes a system call with each addition operation, requesting the kernel's intervention for computation results. Surprisingly, peak consumption in syscall addition is lower than in any other measured scenario. This is due to the sluggish context switch accompanying each system call— involving tasks such as stack pointer copying, interrupt creation, and context restoration— reducing the core's ability to maximize usage for additions, thus reducing current consumption.

Configuration	Number of iterations	Mean time	Std time
IDF	40×10^6	1.19 s	0.23 s
PSF — Kernel	40×10^6	1.21 s	0.13 s
PSF — User	40×10^6	1.20 s	0.13 s
PSF — Syscall	10^6	4.13 s	0.59 s

Table 5.1: Comparison of the time required to compute the addition

Table 5.1 presents the required time for each scenario to complete a full set of additions. One can rapidly see that the time needed to complete the computation is essentially consistent across the first three configurations with small variations, which can be explained by the relatively high standard deviation, with a coefficient of variation ranging between 10% and 20%. However, the syscall configuration takes approximately 3.4 times longer than the other scenarios, despite performing 40 times fewer computations. Hence, the syscall configuration operates at a speed 136 times slower than the others at the same scale. This substantial performance decline stems from the necessity of executing a context switch with each addition operation.

Configuration	Energy	Time
IDF	4.096 Ws	20.24 s
PSF — Kernel	4.180 Ws	20.25 s
PSF — User	4.200 Ws	20.25 s
PSF — Syscall	$40 \times 9.57 = 382.8$ Ws	$40 \times 48.75 = 1950$ s

Table 5.2: Comparison of the time and energy required to compute 9 additions

Table 5.2 contains the energy and time required to compute 9 additions, it shows that those metrics are nearly consistent across all tasks, with only a slight uptick attributable to the increase in FreeRTOS tasks. However, the task with a high volume of system calls significantly surpasses others in resource consumption. Scaling the analysis to accommodate the disparity in calculation scale—1 million for some tasks versus 40 million for the mentioned task—we find that it would have utilized 382.8 Ws and taken 1950 seconds. Hence, it is apparent that the integration of user tasks has minimal impact on energy and time consumption, provided the frequency of system calls remains low.

5.4 Multitasking

In this experiment, we evaluate our implementations of multitasking against the native task scheduling of the IDF, and also evaluate how much our preemptive scheduling implementation helps to improve the overall performance of our solution. Additionally, we benchmark our implementation that permits deep sleeps when idling, thus saving considerable power. In each scenario, we have three tasks; the first one calculates 1 million additions, the second 2 millions and the third 3 millions. In both modes, our scheduling task interrupts them after 2000 ms regardless of their progress. The difference between the two modes, is that in the round-robin mode, the tasks idle until the timer is reached after they are done, whereas in the preemptive mode, the tasks give back the hand to the scheduler as soon as they are done.

5.4.1 Basic Round-Robin

In this mode, while we can see on both configurations (figures 5.8, 5.9) that the CPU spends most of its time idling, it is much more obvious in the PSF , where we can see only ≈ 7 round of computations. Although it can be useful to have idling time to save some battery on the device, in this scenario we want to maximize the number of computations, and therefore we are wasting time while idling, hence our optimization with a preemptive scheduler.

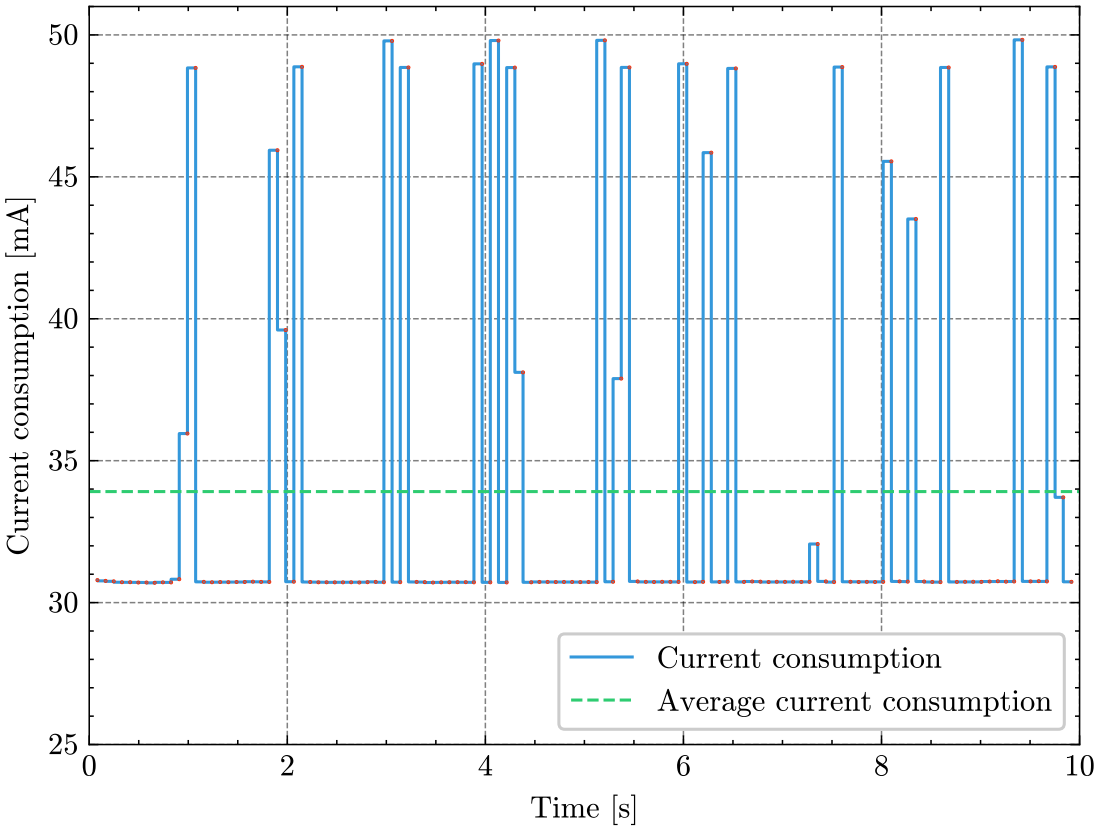

Figure 5.8: Multitasking **without** optimization on IDF

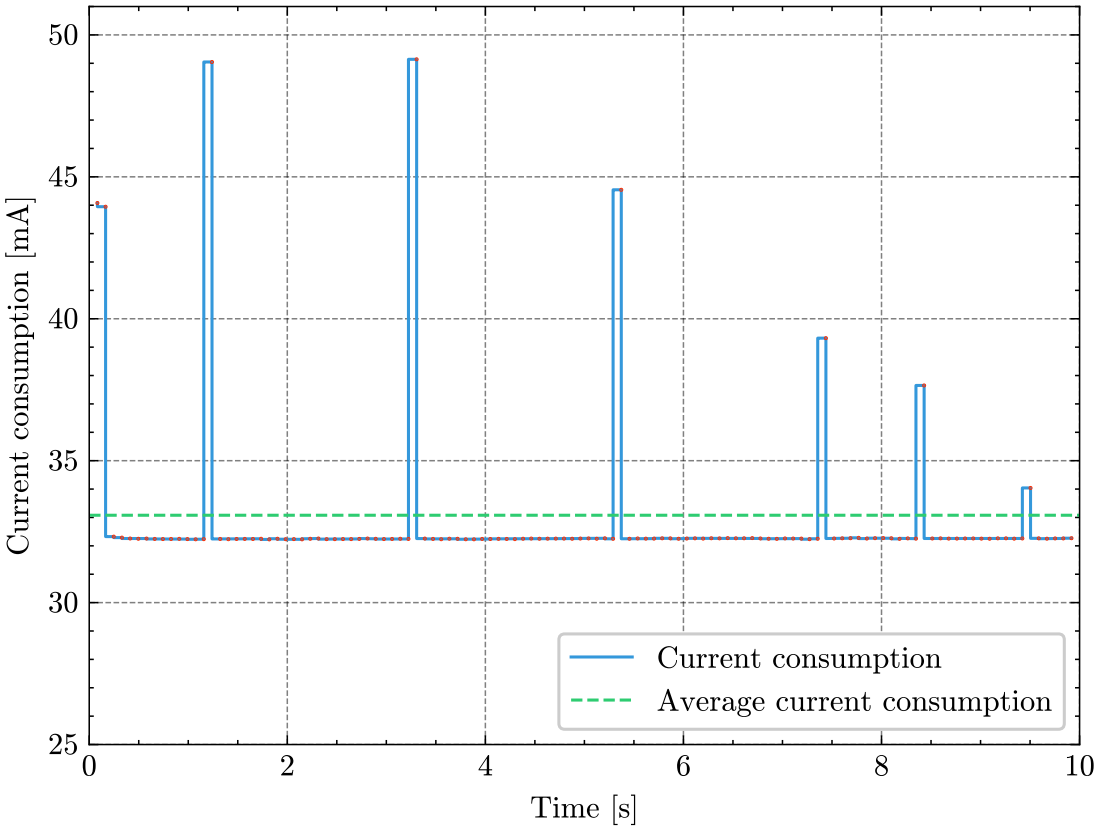

Figure 5.9: Multitasking **without** optimization on PSF

5.4.2 Preemptive Scheduling

The approach underpinning this scheduler, discussed in Section 4.1.1, revolves around promptly handing control back to the scheduler upon task completion, rather than passively awaiting timer expiration and subsequent forced intervention by the scheduler. This method offers several benefits. Foremost, it optimizes CPU usage by reducing idle time. Naturally, without an incentive for timely control relinquishment, users may be reluctant to adopt this approach. However, consider a scenario where each client incurs a fixed fee based on their task's CPU occupancy duration. In such cases, it becomes economically advantageous, especially for short tasks involving data checks and swift termination, to promptly yield control, thereby minimizing resource consumption.

When contrasting with the previous mode, a notable increase in computational throughput is evident on both Figures (5.10, 5.11), with each task executing multiple times per second. Correspondingly, average power consumption has risen,

as expected, due to reduced idle periods.

A closer examination of the IDF (Figure 5.10) and PSF (Figure 5.11) reveals that the latter exhibits even fewer instances of idle time, with occasional dips below 35 mA occurring only when sampling fortuitously captures idle intervals. Once more, the IDF outperforms our implementation, boasting an average consumption of 40.47 mA compared to our 47.87 mA (representing an 18.29% increase), despite similar computational loads.

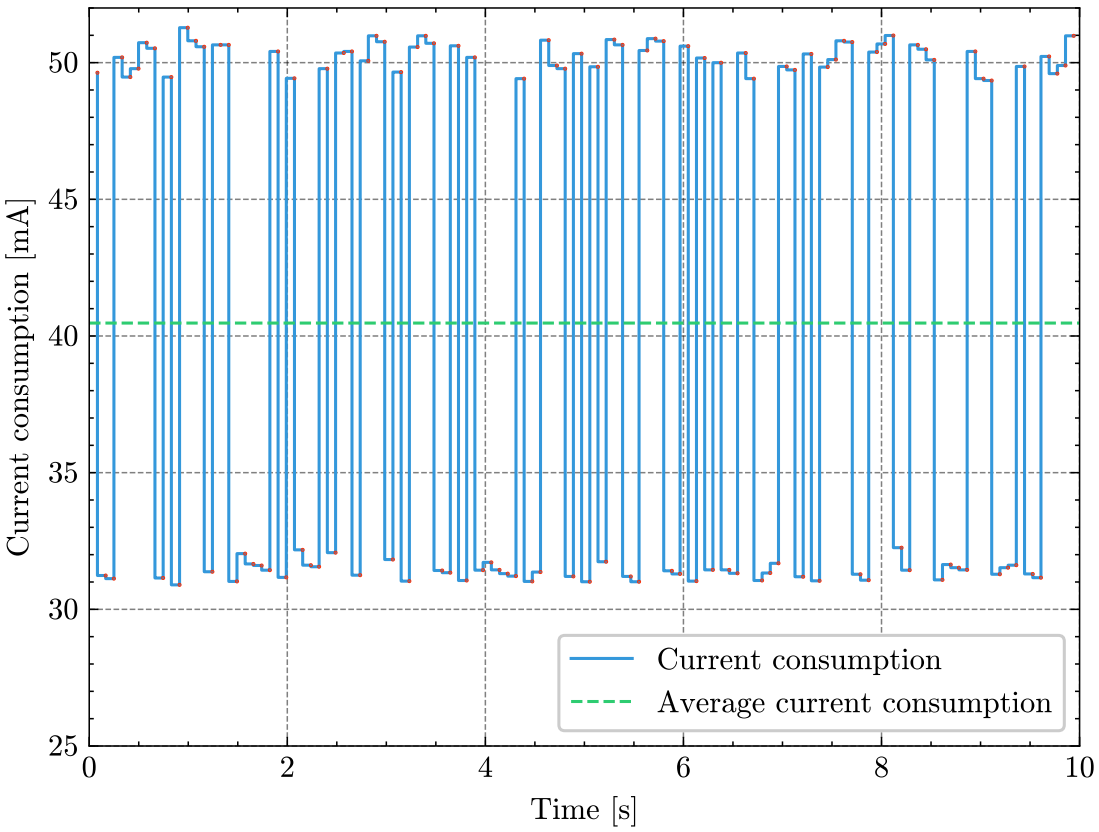

Figure 5.10: Multitasking **with** optimization on IDF

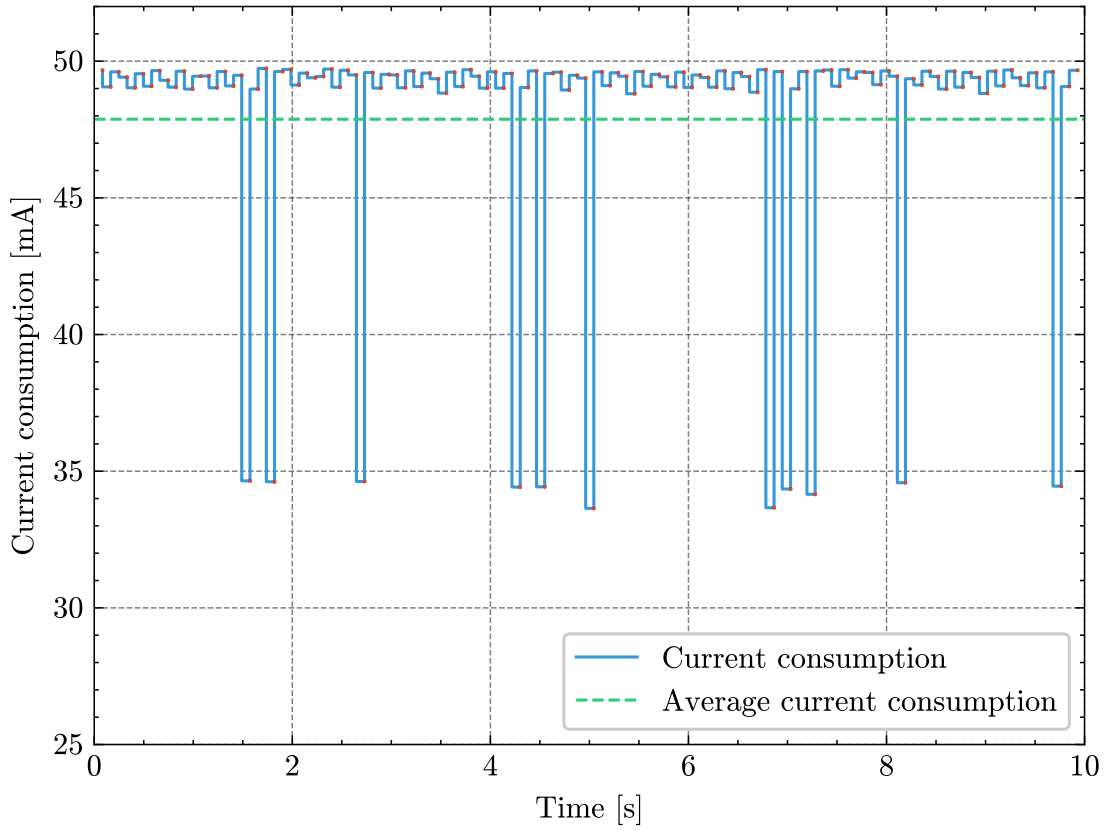

Figure 5.11: Multitasking **with** optimization on PSF

5.4.3 Deep Sleep Optimization

The deep sleep optimization enables the device to go into deep sleep mode, that is, a very low consumption state where the device only listens for an RTC signal to wake up, but on boot up, it has to reinitialize all the device tasks and start as if the device was powered off.

In this scenario, we replicate a use case where each task runs once, then the device enters a deep sleep state for a long period of time (here we chose 5s, for the experiment purpose but in practice it would be much longer), and then an RTC wakes it up and start over. As before, the three tasks compute additions of respectively 1 million, 2 million, and 3 million integers.

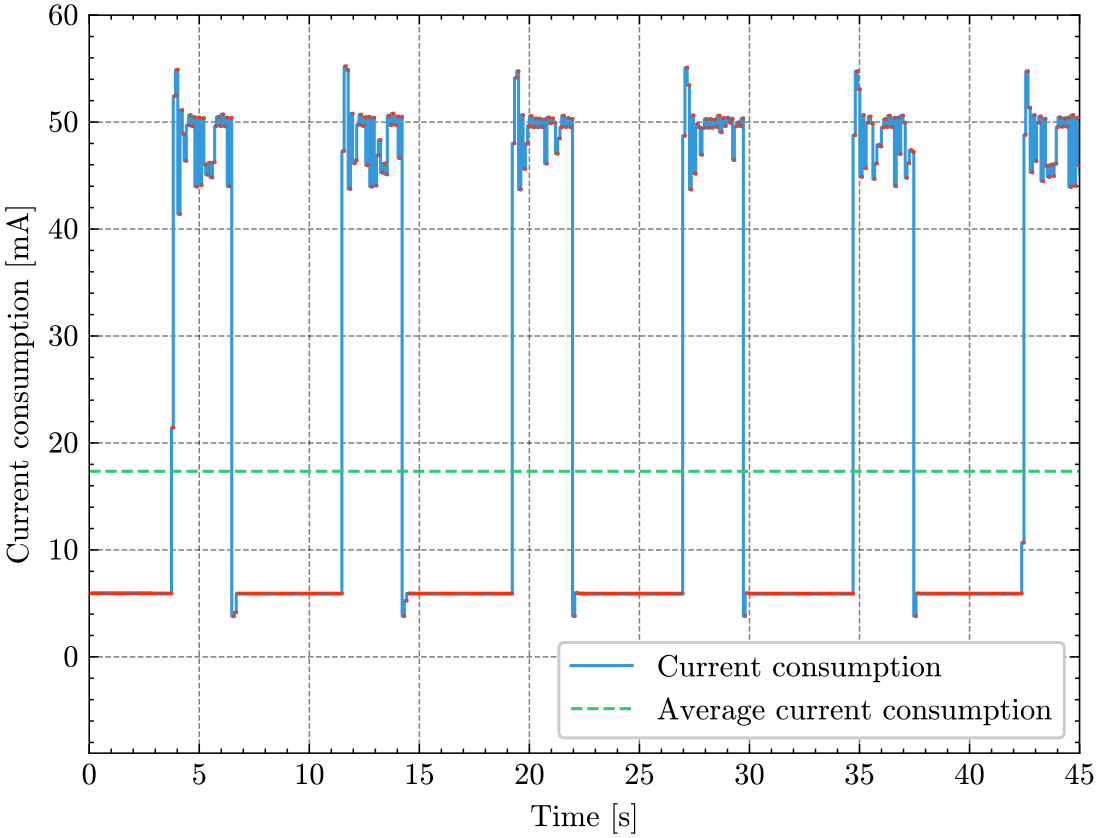

Figure 5.12: Deep sleep with optimization on PSF

Figure 5.12 shows that during sleeping periods, the device's electrical current consumption remains stable at 6 mA. In contrast, the idle current consumption in the previous experiment (Figure 5.9) was approximately 32 mA, indicating an 81.25% reduction in electrical current consumption. According to the measurements done by Espressif [5], by optimizing the hardware of the development board, we should be able to decrease it up to $\approx$ 10 µA. During active periods, we observe that the wake-up process of the device incurs spikes reaching 56 mA, followed by a decrease to around 50 mA. This represents the same current consumption than in previous scenarios.

[5] `https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/api-guides/current-consumption-measurement-modules.html`

5.5 Real World Scenario

The aim of this experiment is to assess the current consumption of an application representative of real-world scenarios. Specifically, we focused on a standard application periodically taking sensor measurements every 500ms and transmitting the data to a cloud server. However, given the variability in sensor consumption based on factors such as origin and usage, we simplified this aspect by having the device transmit dummy values to the server.

In the PSF, the Wi-Fi module is entirely managed by the kernel world, which is responsible for establishing and maintaining the connection to the access point, as well as managing all operations related to layers 1 and 2 of the OSI model. The management of layers 3 and 4 is similarly handled by the kernel world, though this is done through system calls initiated by the user world. The user world, in turn, makes requests to open and monitor sockets and to send packets.

Despite these requests, the actual handling and execution of these tasks is performed by the kernel world. This design choice by Espressif stems from the recognition that the Wi-Fi and the networking modules are sensitive components that requires robust management and security. However, this architecture leads to a significant overhead due to the high frequency of system calls that must be made.

When it comes to the application layer of the OSI model, various protocols can be utilized, including MQTT, HTTP, TLS, and Mbed TLS. For the purposes of our experiment, we opted for a simple HTTP server hosted on a laptop connected to the same access point, as this setup offers ease of use and straightforward implementation. This decision is driven by the need to streamline our testing process while ensuring that our focus remained on the core functionalities and performance metrics of the Wi-Fi module within the framework.

Upon examination, a noticeable uptick in both peak and idle power consumption is observed for both configurations. The idle consumption for IDF (figure 5.13) rises from 30.7mA to 34.8mA, marking an approximate 13.36% increase, while peak consumption consistently exceeds 50mA, often reaching beyond 80mA. The Wi-Fi component emerges as a significant power consumer, whether transmitting data or idling, where it must remain vigilant for incoming network packets without the respite of sleep mode.

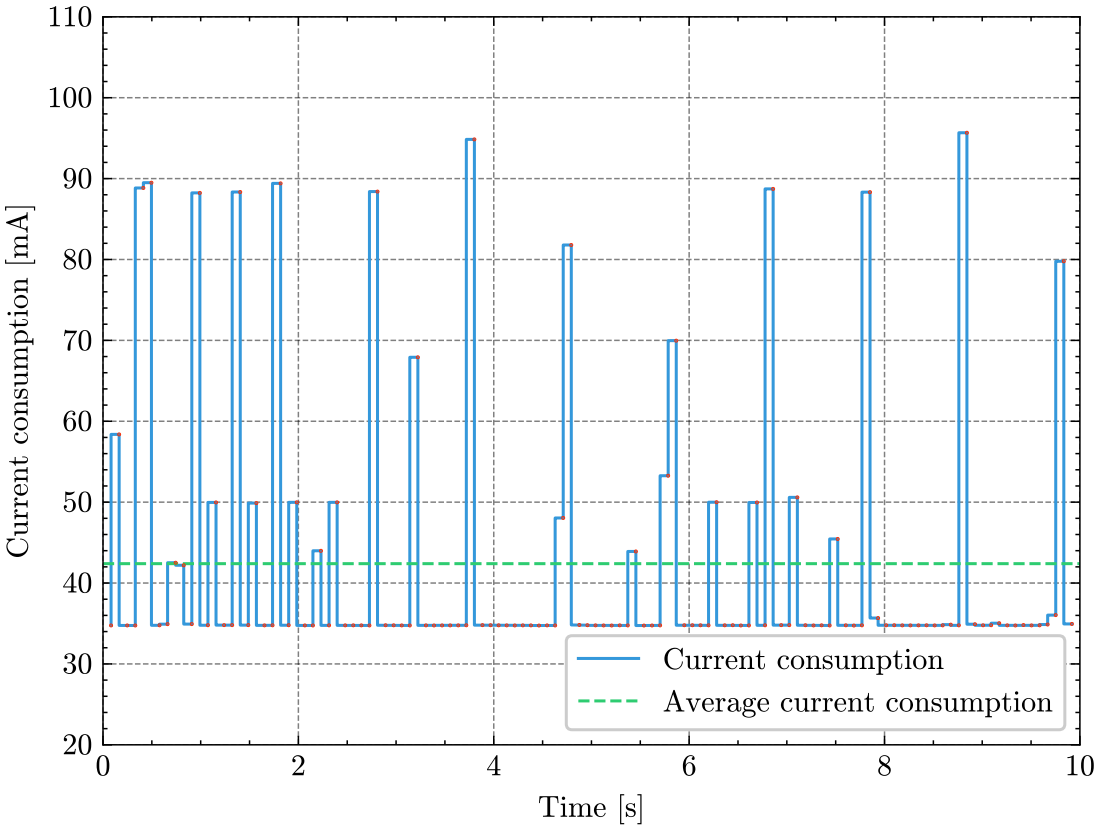

Figure 5.13: Real world scenario on IDF

Similarly, the idle consumption for PSF (Figure 5.14) sees a similar upward trend, climbing from 32.6mA to 36.9mA, equivalent to a 12.84% increase, with peak consumption frequently surpassing 50mA.

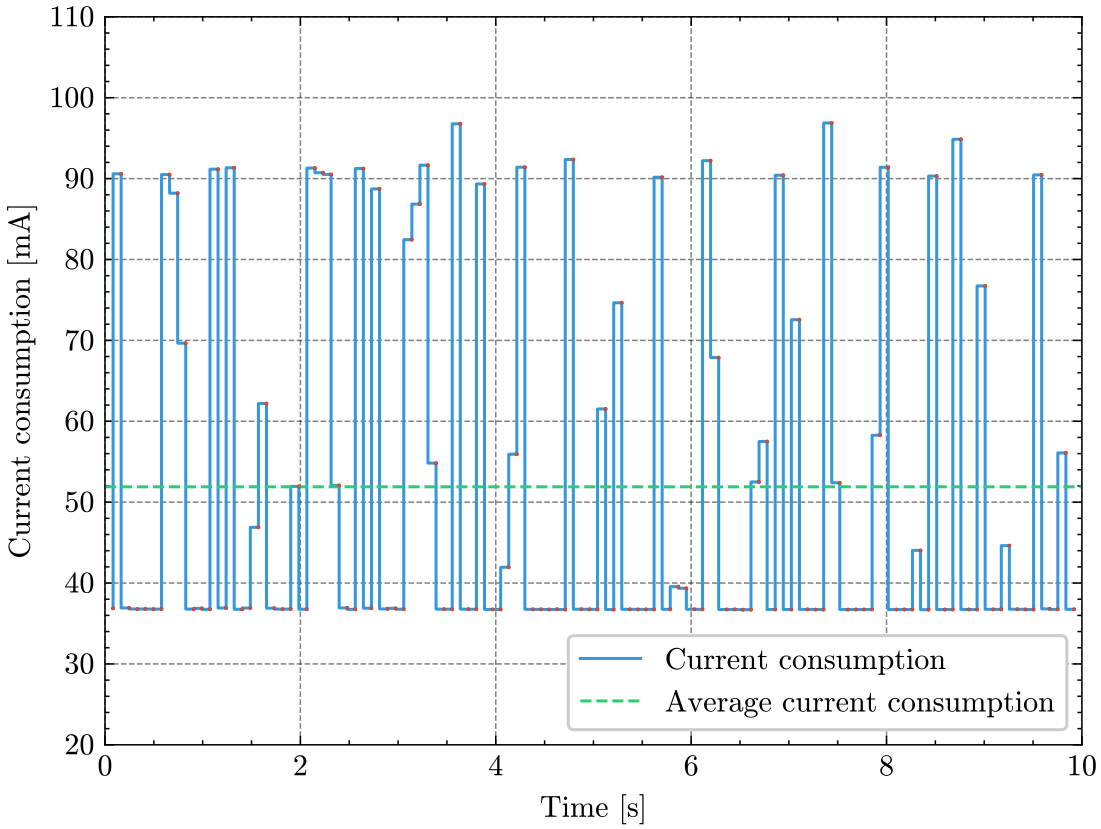

Figure 5.14: Real world scenario on PSF

The, PSF exhibits more frequent and higher peaks compared to IDF, likely attributable to the Wi-Fi module's utilization in the privilege separation framework. The frequent generation of interrupts diminishes overall performance. This disparity is reflected in the averages, with IDF averaging at 42.46mA and PSF at 51.6mA, constituting a considerable 21% increase, notably higher than the previous differences between IDF and PSF.

This result was anticipated, as it consolidates the losses observed in the previous scenarios:

1. **Increased idle consumption:** The idle power consumption is elevated due to the presence of multiple running tasks, including the non-secure world task, the user task, and the dispatcher. In contrast, the IDF version operates without these additional tasks, resulting in lower idle consumption.

2. **Multiple system calls:** The user code necessitates a series of system calls executed in succession. These include retrieving values from the queue, and

performing socket operations such as opening, connecting, writing, and closing. As demonstrated in our cumulative experience, executing multiple system calls in sequence substantially diminishes the code's speed.

While optimizing the code by implementing deep sleep modes and modifying the send request to be completed in a single system call could mitigate these issues, there remains a considerable impact on the battery life of the device. This impact stems from the inherent inefficiencies associated with the multiple system calls and the higher idle consumption due to the additional tasks. Therefore, despite potential optimizations, the overall battery performance is likely to be adversely affected.

5.6 Conclusion

Our findings align with our initial expectations regarding the PSF's impact, indicating a notable but small performance overhead in terms of both power consumption and computational duration during normal usage. The observed increase in current consumption ranges between 6% and 21% across various usage scenarios, a significant factor that warrants careful consideration by system manufacturers and designers.

While it is evident that generating a multitude of interrupts incurs substantial costs, our analysis suggests that diligent application optimization can mitigate these expenses. By investing additional time in application design and strategic planning, it is feasible to curtail the frequency of system calls, thereby alleviating the burden imposed by the PSF. Although the quantifiable benefits may be challenging to measure, the implications for system security and user privacy are profound, potentially resulting in significant cost savings by deterring cyberattacks and minimizing retribution expenditures.

In our exploration of multitask implementation, we observe that the round-robin with preemptive time limit works very well: the isolation between each user task is maintained, the impact on task switches is small, and the user can write their code without taking into account that it will run in an isolated environment.

In conclusion, our solution could provide a suitable solution if the user optimizes the code enough for the use case. Thus, our code must be seen as a framework rather than a "one code fits all solution". The use of PSF on battery powered device is more challenging, as even with optimization, multiple user tasks will have a large impact on the consumption. PSF would be suitable for a battery powered device with only one user application and with the deep sleep enabled.

Chapter 6

Discussion

The work presented in this thesis establishes a robust foundation for secure multitasking in embedded devices, showcasing its theoretical feasibility and practical implementation. Through our research and development efforts, we successfully demonstrate that it is possible to enable multiple users to share a single embedded device while maintaining strict isolation between each user's tasks. This ensures that no individual user is granted full control over the device, thereby enhancing security and preventing unauthorized access or interference among tasks.

However, the potential for further optimization, other types of scheduling, and commercial applications invites extensive opportunities for future exploration and improvement. In this chapter, we will explore the current limitations and present a list of potential amelioration that would greatly improve the usability of this work.

6.1 Limitations

This section addresses two distinct areas of limitations. The first area concerns how our implementation manages errors and recovers from them. The second limitation pertains to deep sleep optimization, which clears task stacks and thereby introduces an overhead.

6.1.1 Error Handling

When a user task attempts to access a restricted memory zone, the PerMiSsion control (PMS)[1]generates an interrupt that is caught by the kernel code. The kernel

[1]This abbreviation is unconventional, yet it is considered the "official" one, as indicated by this forum post: `https://esp32.com/viewtopic.php?t=40040`

can identify the erroneous task and take appropriate measures before resuming all user tasks.

However, as outlined in Section 4.1.3, certain errors, such as a stack overflow, will cause the ESP32 to reboot. To address this, we can disable the offending task that caused the reboot without involving the kernel code. While effective, this approach has significant drawbacks. Firstly, due to the memory being cleared during an ESP32 reboot, we resorted to using NVS to retain the current task and blacklist. This is suboptimal as NVS is slow and has limited write endurance. Secondly, this method does not attempt to restart the user task, effectively banning it indefinitely.

We propose the addition of external volatile memory, such as PSRAM, which would not be cleared upon an ESP32 reboot since the power is maintained. This offers faster access times and greater durability compared to NVS. Furthermore, a more nuanced task banning mechanism could be beneficial. For instance, we could implement a system where a task is banned for exponentially increasing durations using counters, and additionally report the incident to a server for analysis.

Capturing the specific cause of an ESP32 reboot remains a challenge. It is uncertain if this can be achieved without intercepting the serial output from the ESP, which is impractical and would require an additional microcontroller.

6.1.2 Deep Sleep Stack Wipe

The current implementation of deep sleep mode results in the loss of user stacks. This is a typical practice in development, requiring users to be vigilant and modify their code accordingly.

Transferring user stacks to flash memory before entering deep sleep mode would be highly inefficient. Flash memory operations are slow and energy-intensive, and flash memory itself is not designed to withstand a high number of read-write cycles.

Similar to the error handling approach, we propose integrating an external low-power volatile memory module. This memory should remain powered during deep sleep mode, allowing it to preserve user tasks throughout this period.

6.2 Future work

This section outlines various improvements and future iterations of our work, including the addition of a new hardware module to enhance performance that is mentioned in the previous section, a novel method for uploading client code to the device, new scheduling strategies, optimization of context switches between tasks, and a discussion on the development environment.

6.2.1 Hardware

Enhancing the development board could provide significant benefits. Incorporating dedicated PSRAM for storing user application stacks and error handling data can mitigate issues related to memory loss during reboots or sleep modes, while also improving speed and durability. Additionally, integrating external flash memory would facilitate the development of a solution with independent storage for each user application, thereby enhancing flexibility and scalability. Modifying the operating voltage of the development board from 5V to 3.3V would eliminate the need for a step-down converter, leading to increased energy efficiency. Furthermore, removing the component responsible for the COM port would contribute to these energy savings and allow for more precise measurements that closely align with real-world applications. These modifications would not only enhance the board's performance and reliability but also make it more suitable for practical, energy-sensitive environments.

6.2.2 User Code Upload

Our current implementation necessitates that clients transmit their code as source files. These files must then be manually incorporated into the codebase before any OTA updates can be executed. This approach is fraught with several critical issues, primarily related to error susceptibility and confidentiality concerns.

The manual addition of client code to the existing codebase introduces multiple points where errors can occur. Firstly, the process of integrating new code manually is inherently error-prone due to the possibility of human mistakes during the copying and pasting phases. These errors can manifest as syntax errors, incorrect file placements, or even unintentional overwrites of existing code. Each of these mistakes can lead to significant debugging efforts, delayed updates, and potentially malfunctioning features in the deployed application.

From a privacy standpoint, the current method raises substantial concerns. Clients are required to send their source code, which may contain proprietary algorithms, sensitive data handling procedures, or other confidential elements. This necessity to disclose source code could be a deterrent for clients who are protective of their intellectual property. Even with strict security measures in place, the mere transmission and handling of raw source files expose the code to potential vulnerabilities, including interception, unauthorized access, and misuse.

To mitigate these issues, a more streamlined and secure approach would involve enabling clients to compile their code into binary files before transmission. This method would require clients to build their code independently and send only the resulting binary file. These binary files could then be directly unpacked and

integrated into the device's system, allowing for seamless remote updates without necessitating changes to the entire codebase.

Based on our research, implementing this system is feasible but requires modifications to the ESP IDF source code. Since the ESP can only execute code from the internal flash or IRAM, the process must be performed in several stages: first, load the code from external flash memory to ensure sufficient space on the ESP for multiple user applications with OTA capabilities; then, copy the code to DRAM; and finally, transfer the code to IRAM for execution when needed. It will force starting from scratch the OTA system too

6.2.3 Exploring Scheduling Strategies

In this thesis, we developed two straightforward schedulers. The first is the round-robin scheduler, which is advantageous due to its simplicity but tends to waste a significant amount of resources. The second is the preemptive scheduler, which addresses some of this inefficiency by allowing tasks to be interrupted and resumed. However, the preemptive scheduler requires incentivizing users to fully utilize its cooperative capabilities. Beyond these two, many other schedulers could be considered from optimization and commercial perspectives.

From a commercial standpoint, one promising approach is to implement a priority-based scheduler. This system would feature multiple queues corresponding to different priority levels, with clients paying more for higher priority tasks that are scheduled to run more frequently. Additionally, dynamic priorities could be introduced, allowing tasks to request a higher priority when entering critical sections of code that demand swift execution.

From an optimization perspective, particularly in embedded systems with real-time requirements and stringent deadlines, schedulers like Earliest Deadline First (EDF) or Shortest Job First (SJF) are recommended. These algorithms are designed to meet real-time constraints efficiently. However, implementing such optimization techniques poses challenges in terms of fair monetization.

It is important to note that as schedulers become more complex, the dispatcher task experiences increased strain, requiring more CPU time and memory to manage task states, priorities, and deadlines. Therefore, it is crucial for manufacturers to choose a scheduling approach that aligns with their ultimate objectives and the specific demands of their systems.

6.2.4 Smart Copying of Stacks

In the current system, when registering a new task, the client must specify the maximum stack size to the manufacturer. During initialization, the kernel dispatcher task allocates memory for the maximum stack size to ensure that the task can be stored safely when preempted. This conservative approach prevents stack overflow issues. However, the existing mechanism copies the entire stack to a buffer during each context switch, irrespective of the actual stack usage by the task. This practice leads to inefficiencies, as the full stack is copied even if only a small portion is utilized.

To enhance efficiency, we propose an improvement where a task pointer is passed to the dispatcher, allowing the system to copy only the portion of the stack that is actually in use. While this approach has the potential to significantly reduce the amount of data copied, it also necessitates stringent safeguards. Specifically, precautions must be taken to prevent a scenario where a task deliberately uses minimal stack space and then accesses the remaining uninitialized stack area to inspect residual data from previous tasks.

To address this security concern, we recommend copying back only the required stack size and employing the C function `memset` [2] to fill the unused portion of the buffer with null bytes. This ensures that any leftover data from previous tasks is effectively erased, maintaining data integrity and security.

If implemented correctly, this strategy would not only save execution time but also reduce the energy consumption associated with context switches. By minimizing the data copied during each switch, the system can achieve improved performance and efficiency, contributing to overall system optimization.

6.2.5 Tool-Chain and Development Tool

Developers working with user code require a dedicated tool-chain for development, testing, and deployment. The user code should interface with the system call interface, which abstracts the `SYSCALL` assembly instructions, and must be aware of the compatible libraries within the user environment for code creation. Testing, however, necessitates access to the complete code on their own ESP.

One approach is to provide developers with a basic version of the kernel code. However, this would require the ESP owner to consistently update and distribute this framework. A more efficient solution would be to develop an emulator. This emulator would facilitate user code testing and simplify the maintenance of the test environment.

[2] `https://www.man7.org/linux/man-pages/man3/memset.3.html`

Chapter 7

Conclusion

The proliferation of embedded devices, whose number surpasses the global human population, presents both opportunities and challenges. One significant challenge is ensuring the security of these devices, which increasingly handle more data and integrate deeply into our daily lives. Conversely, opportunities abound due to technological advancements that make these devices smaller, more energy-efficient, and more powerful.

This thesis aimed to address these two facets by extending the Privilege Separation Framework developed by Espressif Systems to enable multi-user capabilities for embedded devices while taking into account that embedded devices are constrained by limited memory, computational power, and battery life.

Our primary contribution is the development of a system that allows manufacturers to run different user codes using a scheduler. We designed two types of schedulers: a round-robin scheduler and an advanced preemptive scheduler with cooperative capabilities, allowing tasks to yield control before their allocated time expires.

Building on this foundational contribution, we introduce several optimizations and additional features to enhance the system's production readiness and commercial viability. Key among these is a secure method for handling secrets, enabling the safe use of private cryptographic keys, passwords, and other sensitive data. The partition storing this sensitive data is encrypted and stored in flash memory, preventing manual probing by individuals with physical access to the device.

We also implement a data-sharing pipeline between the kernel and user space, utilizing a queue to facilitate efficient data transfer without spawning new tasks, thereby saving time and reducing power consumption.

Furthermore, we extend our solution to support deep sleep functionality, a crucial

feature for battery-powered embedded devices that must remain operational for extended periods. This functionality allows devices to conserve battery life while remaining capable of waking up in response to external signals.

Upon completing these implementations, we conduct thorough benchmarks comparing our framework and its optimizations against the native ESP IDF framework. These benchmarks reveal that while our framework introduces some overhead, careful application design and optimization can minimize this impact, making the overhead negligible in many use cases.

Lastly, we critically examined our implementations to identify remaining performance issues and limitations. Although our system is not yet ready for industrial deployment, it provides a solid foundation for further development. We also outline potential future enhancements and areas for further research that could build upon our work, paving the way for more secure, efficient, and versatile embedded systems.

Bibliography

[1] Piet De Vaere and Adrian Perrig. "Hey kimya, is my smart speaker spying on me? taking control of sensor privacy through isolation and amnesia". In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023, pp. 2401–2418.

[2] Koen Zandberg et al. "Femto-containers: lightweight virtualization and fault isolation for small software functions on low-power IoT microcontrollers". In: *Proceedings of the 23rd ACM/IFIP International Middleware Conference*. 2022, pp. 161–173.

[3] Tiago Alves. "Trustzone: Integrated hardware and software security". In: *Information Quarterly* 3 (2004), pp. 18–24.

[4] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. "Trusted execution environment: What it is, and what it is not". In: *2015 IEEE Trustcom/BigDataSE/Ispa*. Vol. 1. IEEE. 2015, pp. 57–64.

[5] Sandro Pinto and Nuno Santos. "Demystifying arm trustzone: A comprehensive survey". In: *ACM computing surveys (CSUR)* 51.6 (2019), pp. 1–36.

[6] ARM Limited. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*. Accessed: 2024-05-27. ARM Limited. URL: https://developer.arm.com/documentation/ddi0406/latest.

[7] Sandro Pinto et al. "Towards a lightweight embedded virtualization architecture exploiting arm trustzone". In: *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*. IEEE. 2014, pp. 1–4.

[8] Sandro Pinto et al. "LTZVisor: TrustZone is the Key". In: *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Ed. by Marko Bertogna. Vol. 76. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017, 4:1–4:22. ISBN: 978-3-95977-037-8. DOI: 10.4230/LIPIcs.ECRTS.2017.4. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2017.4.

[9] Sandro Pinto et al. "Towards a TrustZone-assisted hypervisor for real-time embedded systems". In: *IEEE computer architecture letters* 16.2 (2016), pp. 158–161.

[10] Taylor Hardin et al. “Application memory isolation on ultra-Low-power MCUs”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, pp. 127–132.

[11] Amit Levy et al. “Multiprogramming a 64kb computer safely and efficiently”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 234–251.

[12] *ESP32-S3-DevKitC-1 v1.1 - ESP32-S3 - — ESP-IDF Programming Guide latest documentation*. URL: `https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/hw-reference/esp32s3/user-guide-devkitc-1.html` (visited on 05/18/2024).

[13] Richard F Rizzolo et al. “IBM System z9 eFUSE applications and methodology”. In: *IBM Journal of Research and Development* 51.1.2 (2007), pp. 65–75.

[14] “IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices”. In: *IEEE Std 1619-2007* (2008), pp. 1–40. DOI: `10.1109/IEEESTD.2008.4493450`.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**