Criteria for Securing Operating Systems Supporting Low-End Devices in the Internet of Things

by

Yusef Karim

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs

in partial fulfilment of the requirements for the degree of

Master of Computer Science

November, 2021

School of Computer Science

Carleton University

Ottawa, Ontario

Abstract

The Internet of Things (IoT) is now comprised of tens of billions of Internet-connected devices. As the IoT continues to grow in size and complexity, need for specialized IoT operating systems (OSs) becomes increasingly important to facilitate rapid development of secure and portable applications. However, providing such support for the subset of low-end devices called for in resource-constrained environments introduces additional design challenges, particularly with respect to security.

In this thesis, we propose nine criteria to collectively encapsulate several important aspects and considerations for securing OSs supporting low-end devices in the IoT. To begin, we discuss key characteristics, use cases, and OS support for low-end devices in the IoT. For context, we also select and provide a summary of two actively developed IoT OSs with academic origins, RIOT and Tock. We then present our three main contributions, each of which builds upon one another to inform and end in our proposed IoT OS security criteria. First, we review the role of foundational hardware- and software-based mechanisms relating to OS security. We discuss the need for such mechanisms and identify several relating to IoT OSs supporting low-end devices, accompanying each with a case study pertaining to a real-world example. Second, we experimentally examine the use of such mechanisms in both of our selected IoT OSs running on an ARM Cortex-M based low-end device. Finally, we combine these contributions with a literature review to derive, support, and propose nine criteria for securing OSs supporting low-end devices, we further evaluate and compare each proposed criterion against the aforementioned IoT OSs.

Contents

List of Acronyms

6LoWPAN	IPv6 over Low-Power Wireless Personal Area Network
API	Application Programming Interface
BLE	Bluetooth Low Energy
CIA	Confidentiality, Integrity, and Availability
CoAP	Constrained Application Protocol
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
DTLS	Datagram Transport Layer Security
ELF	Executable and Linkable Format
FP	Frame Pointer
HAL	Hardware Abstraction Library
I/O	Input/Output
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IoC	Internet of Computers
IoT	Internet of Things

IPC Inter-Process Communication

IPv6 Internet Protocol version 6

ISA Instruction Set Architecture

LR Link Register

LSB Least Significant Bit

MCU Microcontroller Unit

MMIO Memory-Mapped IO

MMU Memory Management Unit

MPU Memory Protection Unit

NOP No Operation

OS Operating System

OTA Over-the-Air

PC Program Counter

PMP Physical Memory Protection

PPB Private Peripheral Bus

RAM Random-Access Memory

RF Radio Frequency

ROP Return-Oriented Programming

RPL Routing Protocol for Low-Power and Lossy Networks

RTOS Real-Time Operating System

SP Stack Pointer

SSP Stack Smashing Protector

TCP Transmission Control Protocol

UDP User Datagram Protocol

USB Universal Serial Bus

WSN Wireless Sensor Network

List of Figures

List of Tables

List of Listings

Chapter 1.

Introduction

The Internet of Things (IoT), a field characterized by the interconnection of various networked devices (a.k.a., *things*), has seen unprecedented growth primarily due to the smaller physical size, heterogeneity, energy-efficiency, and low-cost of IoT devices [13, 70]. Such characteristics have enabled large amounts of these devices to be purchased and integrated into a variety of applications. The result has been a proliferation of so-called "smart" industries and products, spanning areas such as the home, wearable technology, cities, health care, agriculture, industrial environments, and more [88]. In turn, the number of connected IoT devices has reached tens of billions and continues to rapidly grow each year [182].

The integration of such a vast number of IoT devices would likely be challenging without a heterogeneous [138] set of hardware options supporting different deployment scenarios and use cases within the IoT. Thus, IoT devices range from *high-*

end devices [89] capable of running traditional general-purpose operating systems (OSs) found throughout the Internet of Computers (IoC) [37], such as Linux or BSD variants, all the way down to *low-end* devices only capable of supporting specially designed OSs [70, 179, 88] or *bare-metal*[1] software.

In particular, low-end devices befit a subset of IoT use cases facing constraints, e.g., in cost, physical size, memory, computational power, or energy. These devices are not as capable as higher-end systems with less available storage for code and data. As a result, organizations tasked with developing and promoting technical standards, such as the Internet Engineering Task Force (IETF)[2] and Institute of Electrical and Electronics Engineers (IEEE)[3], have put increased efforts towards standardizing secure and more efficient protocols to minimize overhead for these low-end devices [45, 172, 177, 46, 131, 124, 81].

However, organizations researching and proposing standards, such as the IETF or IEEE, typically do not play any role in integrating these various building blocks together to create secure and usable systems [201]. Further, the large variance in IoT hardware (even when just considering the subset of low-end devices) introduces a need for portability across different architectures [19, 119, 164] and support for unique platform-specific hardware-based security mechanisms. Safety is another aspect of the IoT, since IoT devices typically come equipped with a variety of sensors

[1] Software not relying on OS abstractions or services for any functionality, typically used when trying to satisfy a very specific and restrictive set of constraints or runtime requirements.
[2] https://www.ietf.org/
[3] https://www.ieee.org/

and actuators, they have the ability to directly collect information (sense) from or interact with (actuate) their physical environment. Thus, any additional components or techniques employed to provide security may positively benefit overall safety as well [210]. For example, a system property relevant to both security and safety is that of *fault isolation*, which works towards ensuring failures in one part of the system do not cause other parts to fail as well [104, 34].

Yet, without a standard collection of integrated software components, fragmentation and numerous implementations of not only standardized protocols, but system security principles in general, may occur, which if not carefully done or adhering to security best practices, may lead to security and interoperability issues [9]. In fact, this problem may apply to any given system. For example, in the more established IoC, there is heavy reliance upon the advancement of modern operating systems to meet expected standards of security and usability. As security threats constantly grow and evolve, OSs remain as the standard platform to face such threats while still maintaining usability. In general, OSs provide application developers with fundamental abstractions and interfaces to rapidly develop portable[4] software, in an arguably secure manner. Correspondingly, as the IoT industry continues to grow, so will the need for rapid development of IoT applications, increasing the value of OSs as platforms providing abstractions over and interfaces to necessary functionality broadly applicable to IoT use cases. Coincidentally, opportunity for OSs providing reliable security guarantees may also apply to the subset of IoT use cases best

[4]Portable in the sense that applications may be developed in a similar fashion, even across different device families and computer architectures.

served by low-end devices, but not without challenges imposed by constraints and the consequent limited hardware support of low-end devices.

1.1. Motivation

Worryingly, large-scale botnets, such as Mirai [9] and Hajime [133], made to target high-end IoT devices make it clear that the large IoT attack surface [96, 210, 201] is indeed exploitable. A reasonable concern then is whether (or rather *when*) exploitation of these now network-connected and similarly exploitable [167, 143, 65, 62, 126] low-end devices will occur. Despite this, increasingly complex demands within the IoT are calling for *multi-tenancy*, in which multiple distinct and potentially distrustful executable software components share device resources [73, 213, 104, 3]. Given the resource constraints, multi-tenancy may introduce many security challenges.

Notably, software-based mechanisms and configuration of any available hardware governs access to resources and determines the functionality and security capabilities of a given device. However, some low-end devices may completely lack underlying hardware-based security mechanisms or, when made available, these mechanisms may be not as well understood, particularly by those coming from the IoC. For example, when looking at general-purpose IoC OSs, it becomes apparent that well-established and prevalent OSs, such as Linux and BSD variants, have set the status quo for what security guarantees may be expected and what *foundational* mechanisms are relied upon for such security (albeit these guarantees and expectations are in constant flux).

1.2. Problem

We seek to identify and establish a set of criteria—heuristics, supported by experimentation and research into the current literature—to inform the development and maintenance of secure IoT OSs supporting low-end devices. The IoT is diverse and constantly in flux, as the current number of deployments continues to increase. As entirely new use cases for low-end devices are realized, there is a greater need for inquiry into the security of these devices. The problem then, is how to determine design criteria that: 1) take into account the diverse range of IoT use cases, 2) relate to current and actively developed IoT OSs, 3) consider threats posed by entities within an IoT device's life cycle and, 4) encompass potential need for hardware- or software-based security mechanisms.

In this thesis, we pay particular attention to the last point. As mentioned, low-end IoT devices typically have simpler hardware architectures and lack the same hardware-based mechanisms available to high-end devices. However, in general, certain hardware- and software-based mechanisms may be considered mandatory for upholding important security guarantees in both IoC and IoT devices. We term these mechanisms *security foundations* and discuss them further in Chapter 4.

1.3. Main Contributions

This thesis has three main contributions relating to the security of operating systems supporting low-end IoT devices. Each contribution builds upon the others to gain

high-level insights into security at this three-way intersection of operating systems, Internet of Things, and low-end devices. To our knowledge, there has been little research that focuses on this intersection, we hope that our contributions may serve as a starting point for further research, experimentation, and discussion.

Identification and discussion of security foundations. We analyze what should be considered as security foundations in the development of IoT OSs supporting low-end devices. We identify several hardware- and software-based foundations and give an example for each.

Experimental examination of security foundations. We define a set of experiments for examining the application and utilization of the identified security foundations within IoT OSs. We conduct each experiment on two actively developed open source IoT OSs with academic origins. These experiments progressively provide insight into the security mechanisms used within each examined IoT OS. More importantly, they allow us to examine the role and limitations of security foundations, under an established threat model, in preventing or limiting damage caused by device exploitation.

Establishment of foundational and supporting criteria. Combining the knowledge gained on security foundations with the correlated experimental examination of foundation usage in two open source IoT OSs, we establish four *foundational criteria*. These serve as independent experimentally-derived heuristics towards realizing an initial secure environment for the execution of software components comprising an IoT OS. Finally, through additional research into the current literature, we es-

tablish four additional *supporting criteria*. These are another set of independent, and potentially just as important, heuristics leaning more towards maintaining and supporting the security of IoT OSs which, ideally, already satisfy our proposed foundational criteria. As we will discuss, foundational criteria may require thought early in the design process of an IoT OS, while supporting criteria may be added any time after. Together, they are meant to guide developers towards more secure designs, and provide a considerable starting point for further research and discussion into the security of these systems.

1.4. Thesis Organization

In Chapter 2, we provide general background information on low-end devices, outline the relevance of low-end devices in the IoT through two exemplary use cases, and follow this with information about OSs that support low-end devices in the IoT. This includes preliminary overviews of our two selected OSs, each of which is further scrutinized throughout the paper. Shifting the focus to security, Chapter 3 details our threat model, Chapter 4 identifies and discusses several hardware- and software-based security foundations, while Chapter 5 aims to examine these foundations through a set of progressive experiments conducted across each OS on a low-end device supported by both OSs. Chapter 6 combines the knowledge gained with a literature review to establish our nine foundational and supporting criteria. Finally, in Chapter 7, we conclude our findings and discuss potential future work.

Chapter 2.

Background

This chapter provides background information to facilitate general understanding and provide context before our transition to security-specific content. Section 2.1 provides a more thorough definition of low-end devices, outlining what they consist of and how they may be characterized. Section 2.2 highlights two exemplary use cases in the IoT which evidently benefit from use of low-end devices, but that are of sufficient complexity to potentially warrant OS support. Finally, Section 2.3 gives a general overview of OSs supporting low-end devices in the IoT and introduces two open-source IoT OSs of academic origin, RIOT and Tock, both of which we continually refer to and further examine throughout this thesis.

2.1. What is a Low-End Device?

In general, a *device* consists of both hardware and software. Underlying hardware serves as the primary criteria for classifying what type of software can run on any

given device [138]. In particular, low-end hardware may impose several constraints limiting the amount or capabilities of software targeting such hardware. As mentioned, device constraints are often expressed in terms of cost, physical size, memory (for code/data storage and execution), computational power, or energy consumption [200, 122, 44, 57, 5]. To help categorize the spectrum of constrained devices, the IETF has standardized three device classes [44] based on memory capacity[1].

- Class 0 devices have the smallest resources ($\ll$ 10 KiB of RAM and $\ll$ 100 KiB Flash)
- Class 1 devices have medium-level resources ($\sim$10 KiB of RAM and $\sim$100 KiB Flash)
- Class 2 devices have the most resources ($\sim$50 KiB of RAM and $\sim$250 KiB Flash)

Class 0 devices tend to require specially designed bare-metal software due to severe constraints in memory capacity and processing capabilities [44]. Bare-metal software may rely on minimal abstractions [26, 184], allowing for small, efficient, and custom programs written in suitable low-level or assembly languages. Due to memory constraints, software targeting Class 0 devices typically lack the resources to support a full-fledged OS or securely communicate directly with the Internet, instead software is directly stored then run from Flash memory without any (or a very minimal) supporting runtime environment. Lack of rich OS abstractions (Section 2.3) means these devices typically run monolithic (yet minimal) software targeting very specific

[1]Memory is expressed in kibibytes (KiB = 1024 bytes)

and well-defined use cases.

In contrast, anything significantly above Class 2 may be considered a higher-end (IoT) device subject to fewer constraints [138]. These devices may have enough memory and adequate hardware to support a subset of general-purpose OSs, e.g., custom-built Linux distributions [47, 215]. These devices may use well-established hardware-based security foundations and networking stacks widely available to commodity systems (desktops, laptops, mobile phones, etc) to provide security guarantees and necessary Internet-communication, making them relevant and able to adequately satisfy use cases within the IoT facing fewer constraints [37, 89].

2.1.1. Characterizing Low-End IoT Devices

Although devices categorized as Class 0 or those significantly above Class 2 will play a role within the IoT [138], broad applicability and particular use cases may be better served by devices categorized as Class 1 and Class 2 (Section 2.2). We will roughly focus on devices fitting within Class 1 and Class 2. However, the memory capacity constraints that define such classes are only one of the many aspects we must consider when dealing with low-end IoT devices. Particularly because memory capacity is constantly in flux. Memory constraints may provide considerable challenges for some use cases, less to others, or simply be alleviated in time as technology progresses. Thus, additional constraints, such as energy, must also be considered when taking into account the many IoT use cases that exist, some of which may have finite

or intermittent availability to energy [4].

It is worth noting that constraints are not a new problem in computing. The history of operating systems, and the hardware they were built to support, dates back to the mid-1950s with early mainframes—expensive room-sized computers requiring specially trained staff to operate—and progressed rapidly towards the OS-supported phones, tablets, and personal computers we have come accustomed to today [187, p.7-38], many of which are interconnected and form the IoC. These systems undoubtedly faced similar constraints to what we have discussed thus far, especially in the earlier days. However, they were likely never considered as low-end or found to suffer from many constraints together. A mainframe may have had memory or computational power constraints, yet it was the size of a room and demanded unreasonable amounts of energy to perform.

The term low-end comes out of the necessity to satisfy particular use cases. The need for small, energy-efficient, and potentially cheap devices, in turn, drives the need for specialized low-end devices able to satisfy such constraints. In this section, we characterize low-end IoT devices by highlighting several aspects and considering them together. Every aspect discussed below may not be wholly unique to low-end IoT devices, however, together they highlight considerations and challenges for the current and future development and support of such devices in the IoT.

Microcontroller. Definitively, these devices are typically based on simple microcontroller units (MCUs). Heterogeneous in nature, MCUs range from 8-bit MCUs

(e.g., Atmel AVR [119]) all the way up to a variety of 32-bit MCUs (e.g., ARM Cortex-M [22] or RISC-V variants [164, p.33]). Any given MCU may consist of a central processing unit (CPU) (a.k.a., *core*), integrated with memory, input/output (I/O) and security-related peripherals, and secondary, yet tightly integrated[2], chips for remote radio frequency (RF) communications. As depicted in Figure 2.1, all the aforementioned components will communicate with one another through a set of internal shared bus protocols[3]. Initial software may be uploaded to Flash memory storage through physical means by connecting and uploading compiled code over an exposed debug port, e.g., via communication over the JTAG or SWD debug protocols [17, 82].

[2]Due to the integrated nature of MCUs, some may instead refer to them as system on a chips (SoCs) [100], particularly when referring to those integrated with secondary chips. For our purposes, MCU and SoC are interchangeable; we remain consistent in this thesis by using the term MCU.

[3]Many high-end systems use highly extensible external buses, such as PCIe [84], to integrate necessary components, e.g., non-volatile and volatile storage. Internal buses found on MCUs are typically much simpler. A common standardized example of this is the Advanced Microcontroller Bus Architecture (AMBA) [15, 16], commonly used in ARM MCUs [18, 22].

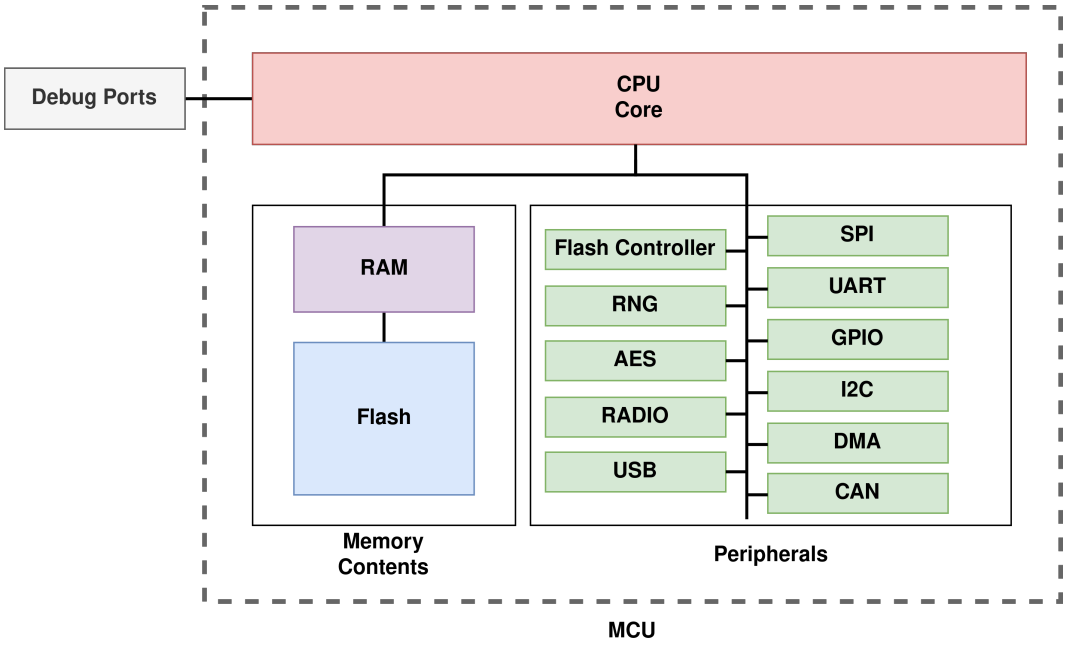

Figure 2.1: MCU hardware overview

Radio. Specific to IoT, low-end devices typically must have some means for RF communications themselves. Depending on the selected MCU, a RF radio may either be integrated and directly accessible over its shared peripheral bus [135, 24] or provided as an external hardware component [134, 175] connected to the MCU and communicated with via a physical peripheral connection, e.g., UART, SPI, or I2C.

Constraints. As mentioned, Class 1 and Class 2 devices may be limited in memory, i.e., persistent Flash for code storage and non-persistent Random-Access Memory (RAM) for data storage. The constraints imposed by memory limitations may vary depending on the use case, but nonetheless point out that developed software must work within hardware constraints, such as memory. However, with respect to memory, these devices have enough memory and processing capabilities to support

specially designed, yet complex software, such as multiple networking stacks and protocols [30, 79, 201] with the potential to communicate directly with the Internet [127]. Thus, despite hardware limitations, the applicability of low-end devices may still coincide with scenarios of non-trivial complexity and impose challenges on software design. Moreover, certain use cases may involve long-lived battery operations [122, 200, 57, 5] or integration into physically confined areas [136, 48, 56]. Thus, constraints, particularly in hardware, energy consumption, and size (all of which additionally relate to constraints such as cost and computational power), are central to how we characterize low-end devices.

Single Physical Address Space. Consequently, constraints require low-end devices maintain simple memory hierarchies and forgo complex memory management (Section 4.1). An example of a resulting memory layout is depicted in Figure 2.2, where all code stored in Flash memory and program data stored in RAM simply share the same single physical address space. Further, for software to directly access peripherals (including those used to interact with the physical world), peripherals are mapped to the same physical address space as memory-mapped IO (MMIO). We will discuss throughout this thesis the consequences of a single physical address space, particularly with respect to the growing complexity in the IoT.

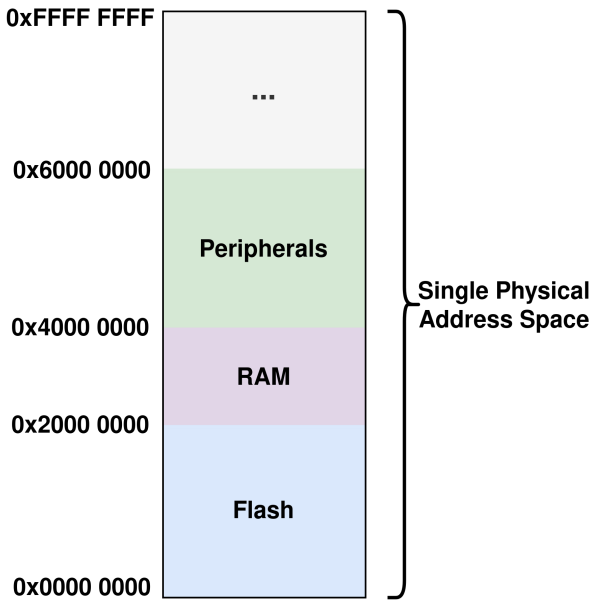

Figure 2.2: Simplified MCU memory layout

Increased Developmental Cost. Potential demand for varied and increasingly complex use cases within the IoT (Section 2.2) may result in demand for and increased complexity of supporting software (and potentially hardware), as similarly seen with the IoC. Efficiently and securely designing these systems now takes considerable effort, care, and expertise. For example, devices may require correct dynamic management of low-power modes [5, p.13-14][49, 132] if constrained in energy, wireless protocols operating at different frequencies for short, medium, or long range communications [77, 201], or software satisfying a set of criteria (Chapter 6) in order to define, establish, and maintain system security. Later discussed (Section 2.3), Class 1 and Class 2 contain adequate memory to support specially designed IoT OSs [70, 179, 88], such support may prove increasingly important for carefully considering constraints and decreasing developmental costs as complexity in software

and hardware increases.

2.2. Exemplary Use Cases of Low-End Devices in the IoT

This section reviews two different IoT use cases benefiting from or requiring use of low-end devices. These use cases exemplify various constraints, highlight specific demands for complex software components, and consequently provide subtle motivation and context relating to the development, use, and security of OSs supporting low-end devices in the IoT.

2.2.1. Fitness Watch

Fitness watches are devices users may wear daily to track their fitness and health-related activities. A fitness watch may contain several sensors to collect information, and actuators, such as small motors, to alert users. Typically, fitness watches do not connect directly to the Internet, instead, they make remote connections to nearby devices via low-power network protocols such as Bluetooth Low Energy (BLE) [41]. For example, a watch may connect to a more powerful mobile device (acting as a gateway) to have collected data forwarded to a corresponding cloud service on the Internet [98, 56]. Additionally, the fitness watch may receive data/notifications from a cloud service via the smartphone [77]. Due to battery operation, power consumption and efficiency must be considered when selecting hardware and developing software for smart watches, as displaying rich user interfaces, performing large amounts of pro-

cessing, and transmitting data all require energy. Additionally, for fitness watches to be lightweight and comfortably fit on a user's wrist, the integration of selected sensors, batteries, and processing units may face physical size constraints depending on the design.

Figure 2.3: Fitness watches

For the reasons above, low-end IoT devices may be deemed a reasonable choice in the design of a fitness watch. In fact, a teardown [63] of a Fitbit[4] Flex (top watch in Figure 2.3) reveals its main processing unit is a low-power ARM-based Cortex-M3 [18] MCU [23] fitting directly between Class 1 and Class 2 (Section 2.1.1) connected to an on-board BLE hardware module. Similarly, looking at a teardown [59] for more complex device by Fitbit, the Ionic (bottom watch in Figure 2.3) is similarly supported by a low-power (yet more feature rich) ARM-based Cortex-M4 [22] MCU [199], slightly above Class 2 in terms of memory, but evidently facing similar

[4] https://www.fitbit.com

constraints in energy as any other fitness watch. Interestingly, the Ionic provides a rich graphical user interface (GUI) and app store running atop the specially designed proprietary Fitbit OS[5]. Applications[6] may be provided by third-parties and, due to prior discussion (Section 2.1.1), we may assume[7] run within a single physical address space once loaded by the OS.

For these simpler and cheaper fitness-oriented devices, MCU-based hardware may play a role in enabling several days worth of usage before recharging the battery [60, p.8][61, p.8]. In contrast, *smart watches* provide feature-rich functionality and typically support tailored versions of well-known general-purpose OSs requiring complex hardware but, for example, have shorter battery life[8]. However, as displayed by the Ionic watch, low-end IoT devices, like fitness watches, are becoming more complex. Supporting multiple applications on simple MCU-supported devices introduces complexity and always-on network connections, BLE in this case, introduces potential for adversarial interactions [203]. Abstractions provided by an OS (Section 2.3) may help reduce the complexity of developing such systems, yet the foundational means for securing such OSs or criteria to help guide secure development have not been studied as thoroughly compared to general-purpose OSs.

[5] https://www.fitbit.com/technology/fitbit-os
[6] https://gallery.fitbit.com/apps
[7] Fitbit OS being completely proprietary means we can not easily confirm our assumptions first-hand.
[8] For example, the Apple watch only supports 6–18 hours of usage [11].

2.2.2. Wireless Sensor Network Node

Wireless sensor networks (WSNs) are comprised of large numbers of interconnected *nodes* utilizing one or more sensors to monitor surrounding physical and environmental conditions, e.g., vibration, temperature, sound, light, etc. WSN nodes may be similarly categorized as low-end devices [94]; they are equipped with wireless transceivers and typically distinguished by their low-cost, low-power, scarce memory capacity, and basic processing capabilities making the networks they form broadly applicable in a variety of applications, such as home and building automation, agriculture, health care services, smart grids, and more [219].

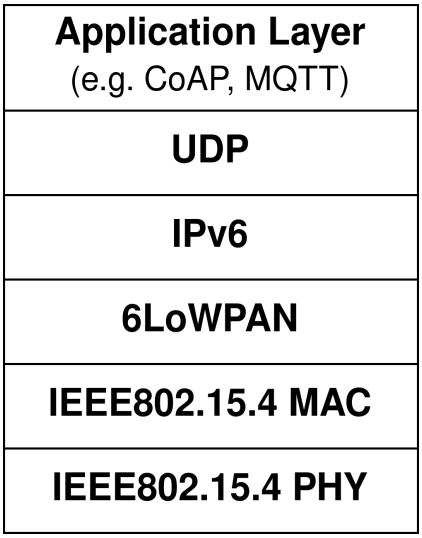

Figure 2.4: Example 6LoWPAN network stack overview

Interestingly, WSNs based on IPv6 over Low-Power Wireless Personal Area Network (6LoWPAN) [101] technologies and standards have an adaption layer allowing them to directly connect to the Internet (and hence partake in the greater IoT) [95].

Shown in Figure 2.4, a 6LoWPAN networking stack is fundamentally supported by the IEEE802.15.4 standard [81] defining the physical (PHY) layer and medium access control (MAC) layer for these low-end wireless devices. The 6LoWPAN adaption layer [123] enables the transmission of IPv6 [53] packets over an IEEE802.15.4 link, primarily providing header compression, fragmentation and reassembly, and stateless autoconfiguration (as outlined in RFC 4944). Consequently, 6LoWPAN devices may communicate directly with the Internet using standard IP-based protocols such as the User Datagram Protocol (UDP) [146]. IEEE802.15.4 and 6LoWPAN along with other protocols, such as the IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL) [209] or Constrained Application Protocol (CoAP) [177], provide efficient, practical, and standardized means for connecting devices to the Internet that would otherwise be too constrained for using conventional networking protocols.

Developing a full-fledged 6LoWPAN networking stack (or any networking stack in general) is not an easy task, especially given constraints and highly heterogeneous low-end hardware. Evidently, many individuals researching 6LoWPAN rely on implementations primarily provided by IoT OSs [95, 219, 214] for experimentation, IoT OSs potentially serve as platforms with established abstractions and implementations. For these same reasons, IoT OSs serve as good platforms for developing real-world 6LoWPAN applications, however, the real-world brings potential for adversarial entities and interactions (Chapter 3), thus, security may be an important requirement for such applications (Section 6.7).

In addition to security, the deployability and energy consumption of WSNs are

important aspects to consider [3, 30]. Developing devices or platforms [3] supporting multiple applications sharing resources may reduce the deployment burden and cost of experimentation, yet introduces the aforementioned challenges of dealing with multi-tenancy. Moreover, the most scarce resource within WSNs—energy—may require notable considerations by software providing runtime environments (e.g., IoT OSs) for 6LoWPAN-based WSNs, especially for WSNs solely reliant on energy-harvesting methods for power [4]. Providing means to determine energy used for a given task on an IoT device may be a novel feature provided by software underpinning the runtime of such applications, a feature potentially easier achieved by systems providing distinct (i.e., multi-tenant) fully mediated applications with the capability of isolating and monitoring energy consumption [2].

2.3. Operating Systems Supporting Low-End Devices in the IoT

Operating systems may avoid redundant developments and maintenance costs for low-end devices within the IoT, particularly if the OS can broadly satisfy a majority of use cases in a secure and efficient manner. However, to achieve broad adoption several considerations must be made during the initial and continued development of such systems, some of which we highlight below:

Resource Constraints. Evidently, due to the memory-constrained nature of low-end IoT devices, a small memory footprint is required of the OS to fit on the device and provide enough additional space for user applications. Further, certain use cases

may be constrained by limited energy sources, thus, will require an energy-efficient OS, for example, one using available deep sleep modes or optimized preemptive task scheduling to reduce power consumption.

Limited Hardware Support. As a consequence of the aforementioned resource constraints, low-end devices forgo complex hardware. As discussed, the most prominent consequence of this is the single physical address space, the memory space that all software, including that of the OS, exists in, shares, and can directly access (in the absence of any preventative mechanisms). OSs supporting low-end devices can not rely on the same hardware-based foundations for providing security; achieving similar or improving upon the security guarantees commonly found within IoC will always be met by limited hardware support. Certain MCUs may simply lack any alternative hardware-based security mechanisms altogether, design goals thus must take into account what can be realistically achieved, both through available hardware and in software.

Differing Uses Cases. If being developed to support a broad range of IoT uses cases pertaining to low-end devices, OSs may need to consider supporting a diverse set of low-end hardware. The vast selection of *commercial off-the-shelf* (COTS) hardware—hardware typically publicly documented and easily available for purchase—alone provides a wide variety of options, each variably addressing trade-offs between satisfying constraints or offering additional hardware-based functionality. In turn, OS developers must consider these trade-offs and how they may affect functionality, efficiency, security, and applicability to differing uses cases.

Additionally, timely and deterministic execution (i.e., known worst-case execution times and interrupt latency) between software tasks may be mandatory for certain use cases. Thus, if aiming towards wider adoption, an IoT OS may benefit from providing deterministic runtime guarantees, commonly referred to as being a *real-time operating system* (RTOS).

Finally, differing use cases may require different communication protocols. As a result, OSs may need to support several networking stacks for seamless network connectivity through a variety of protocols [70].

Multi-Tenancy. Multi-tenancy is where multiple applications may be independently developed then loaded onto a device for execution. Although not every IoT use case will call for multi-tenancy, those that do, which rely on or require low-end devices, may face barriers leading to redundancy in design, software development, and network maintenance, if made to only support one dedicated software application [213, p.3-7].

Referring back to Section 2.2, we learned, whether it was to allow users to download third-party applications from an app store, reduce deployment burdens and costs, or determine energy usage of distinct tasks, that each use case could benefit from multi-tenancy. Yet, multi-tenancy may greatly affect OS design requirements, especially with respect to security [73] and particularly in the face of constraints and limited hardware support.

2.3.1. General Overview

Given these considerations, discussion on how IoT OSs may protect themselves and the applications they serve will be important, especially before any one of them becomes widely adopted. However, before focusing purely on security, we must first understand the components and abstractions IoT OSs need and may provide for general functionality and fundamental operations.

In any case, execution of an IoT OS begins within the *kernel*. The kernel defines important system properties regarding modularity, scheduling, programming model, memory allocation, supported programming languages, and provided abstractions [70]. For example, the *programming model* defines how application developers model their program, typically with respect to concurrency and inter-process communication (IPC). Generically depicted in Figure 2.5, the primary set of software components comprising an IoT OS kernel may be considered to be self-contained building blocks. Many of these blocks may build upon the ones below them, defining additional functionality or providing abstractions to be used by user applications in the end.

Overall, kernel design choices directly impact the overall functionality, capability, flexibility, and security of the entire system. However, as software is confined to a single physical address space, the boundary between user applications and the kernel may be less distinct. Nonetheless, abstractions provided by the kernel may be realized as an application programming interface (API), a set of methods either

directly invokable or requiring a hardware-based context switch to be used. Below we give a brief overview of likely software components and abstractions provided, for general functionality, by an IoT OS kernel.

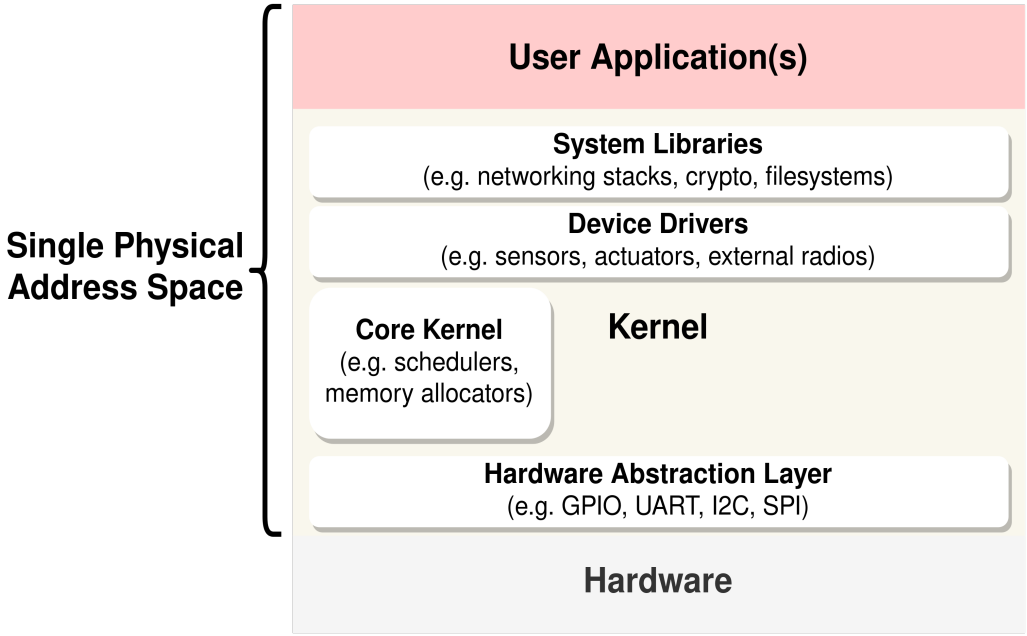

Figure 2.5: Generic layout of an OS for low-end IoT devices

Hardware Abstraction Layer. The *hardware abstraction layer* (HAL) serves to abstract the underlying hardware by providing a well-defined and portable interface over various MCUs, including the CPU core, interrupt handling functionality, and common peripherals (e.g., GPIO, UART, I2C, etc). In many cases, developers will start development on commercial off-the-shelf development boards[9], thus, the HAL may include support for commonly purchased boards containing extra on-board hardware (e.g., LEDs, buttons, etc).

[9]See Figure 5.1 for one such example.

Core Kernel. Potentially the most important design decision for any OS. The *core kernel* can be designed using several different architectures; examples such as the exokernel, microkernel, monolithic, or hybrid approach [70] all provide different functionality, each with their own advantages or disadvantages. For example, the microkernel approach aims to provide a minimal set of features (e.g., scheduling and memory allocation) in the core kernel, then allow for greater space (i.e., program memory) and flexibility for developers to extend this functionality (e.g., through additional device drivers or system libraries). Interestingly, microkernels may provide several security benefits [40, 74], primarily due to separation of privileges (supported by hardware-based mechanisms). However, within the heterogeneous set of IoT devices, some simple devices, e.g., those based on 8-bit AVR MCUs [119], lack multiple privilege modes (and means to subsequently enforce isolation). Thus, microkernel approaches may not gain the equivalent security benefits depending on OS implementation and supported hardware. We discuss this further in Chapter 4.

The core kernel must also provide some way for applications and drivers to be loaded then scheduled. This, in turn, defines the primary concurrency model of the OS and greatly impacts important system properties, such as energy efficiency, real-time capabilities, and security.

Device Drivers. Each external device, e.g., sensor, actuator, or external radio may communicate over MCU peripherals abstracted by the HAL. However, each external device may require (vendor-)specific setup commands and subsequent communication flows. *Device drivers* provide easy-to-use and MCU agnostic interfaces to external

devices, typically relying on HAL constructs for underlying communication.

System Libraries. Developed atop the aforementioned components, *system libraries* implement additional useful functionality. This may include entire networking stacks implementing various protocols, such as link layer protocols (e.g., IEEE802.15.4, BLE) and application layer protocols (e.g., DHCP, DNS, CoAP). Additionally, libraries may include cryptography primitives, filesystems, and supporting utilities for application developers to utilize when building their IoT systems.

Overall, one of the primary goals of an IoT OS is to equip users with the tools and interfaces for rapid and portable development of applications targeting low-end devices. Ideally, developers should be able to load multiple independent applications onto the same device and have guarantees that, upon execution, each application will not greatly affect the others, unless explicitly specified. We will discuss this further, after we establish a threat model (Chapter 3) for these low-end IoT devices.

2.3.2. RIOT Overview

RIOT [31] is an open source OS [158], actively developed by a growing world-wide community. It aims to minimize resource usage, support a very broad range of (8-bit, 16-bit, and 32-bit) MCUs [33], reduce code duplication across configurations, provide code portability across supported hardware, be easy-to-program, and provide soft real-time capabilities [31]. RIOT is implemented in the C programming language and mainly supports applications written in C or C++.

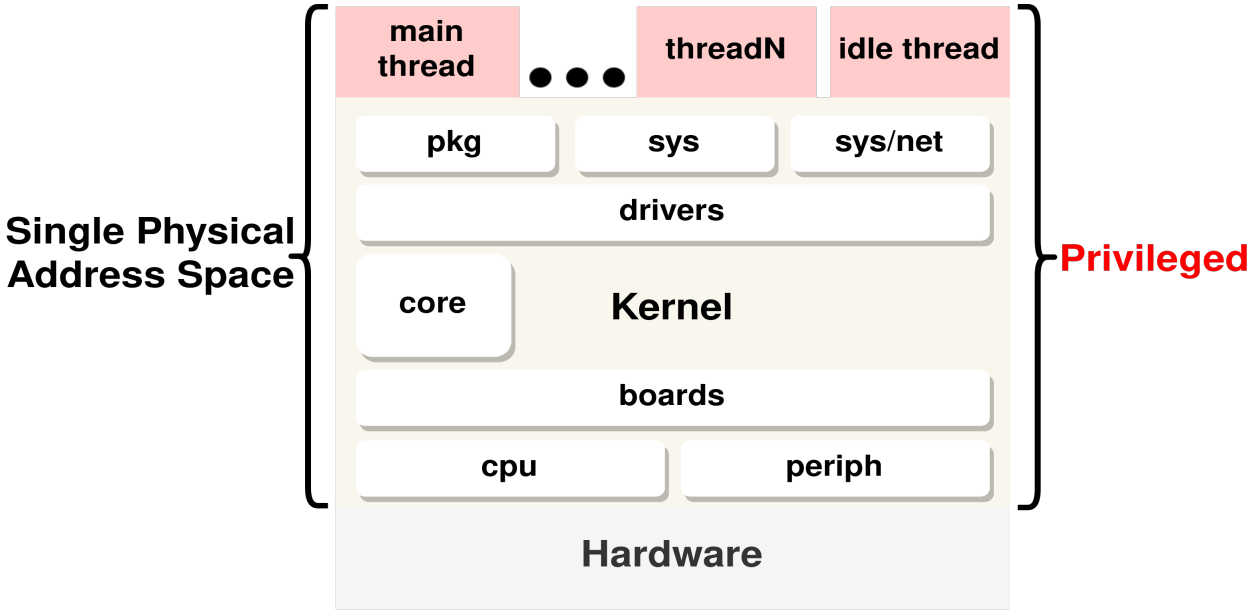

Figure 2.6: RIOT OS overview

RIOT can be thought about using self-contained building blocks [31], as seen in Figure 2.6. Starting with the hardware-dependent blocks, `cpu` relates to supported MCU families (e.g., ATmega, STM32, nRF52), providing functionality to support startup, power management, interrupt handling, and context switching [156]. Within each supported MCU family, `periph` provides the hardware abstractions for the variety of supported peripherals (e.g., GPIO, I2C, etc). Finally, `boards` selects, configures and maps the target MCU, defining the physical devices RIOT supports in general.

RIOT's kernel is developed in a modular fashion. The `core` implements important kernel functionality (e.g., schedulers, memory allocation, etc) and basic data structures. Through `core`, RIOT provides an efficient, but optional, multi-threading pro-

gramming model, whereby, each user-defined *task* is run in its own thread and given a priority value on creation (higher values mean lower priority). Concurrency primitives such as mutexes, semaphores, and message queues provided by the kernel enable inter-process communication and synchronization between tasks. Further, based off of these fixed task priorities, RIOT employs a *tickless* (i.e., interrupt triggered and not dependent on periodic system timer ticks) scheduler, allowing for soft real-time capabilities via deterministic context switching between threads. Low-priority tasks can be preempted in order to deal with higher-priority tasks or interrupts [31]. If there are no pending events, the RIOT kernel will automatically switch to an *idle thread*, with the lowest priority, putting the MCU into the deepest possible sleep state to reduce energy consumption.

```c
#include <thread.h>

char stack[THREAD_STACKSIZE_MAIN];

void *thread_handler(void *arg) {
  (void)arg;
  puts("In another thread!");
  return NULL;
}

// User application begins here, starting in the main thread
int main(void) {
  puts("In main thread!");
  thread_create(stack,         // Start of preallocated stack memory
                sizeof(stack), // Size of the thread's stack in bytes
                THREAD_PRIORITY_MAIN - 1, // Priority of the new thread
                0,                        // Optional flags (omitted)
                thread_handler, // Pointer to code executed in new thread
                NULL,           // Argument to the function
                "thread");      // Human readable thread descriptor
  puts("Continuing in main thread!");
  return 0;
}
```

Listing 2.1: Creating a thread in RIOT

The `drivers` block includes device drivers implemented separately from the core kernel. One or more of these software modules can be optionally included along with the core kernel, typically providing a high-level abstracted interface to components connected or communicating with the MCU, such as sensors, actuators, or network transceivers [31]. Importantly, they are independent of underlying hardware, can be instantiated more than once, and typically aim to provide a high-level API for easy interaction.

RIOT uses static memory allocation (i.e., allocation only on the stack) within the kernel, allowing it to meet stricter timing requirements through constant time kernel tasks (e.g., scheduling or handling inter-process communication). However, it is possible for drivers to make use of dynamic memory allocation, but this is heavily avoided by RIOT developers, as outlined in their coding conventions [155]. In contrast, dynamic memory allocation methods are provided for users to freely use in their application code. [70].

Finally, `sys`, `pkg`, and `sys/net` provide system libraries (e.g., cryptography, file systems, etc), additional third-party components, and networking components respectively. Application developers use these libraries to implement their primary IoT-related logic/tasks.

2.3.3. Tock Overview

Initially proposed in 2014 [142], Tock [104] is an open source [198] OS actively developed by a growing community of contributors. It aims to provide concurrency, memory efficiency, dependability, fault isolation, and the ability to dynamically load applications [104]. Implemented in the Rust programming language [117], the Tock kernel uses Rust-specific features (Section 4.2.2) to gain compile-time memory safety guarantees and develop powerful abstractions over various hardware and software components. Tock supports user applications written in a number of supported languages, leveraging hardware-based mechanisms (Section 4.1.1) found on modern

MCUs to isolate each individual application. Due to its hardware requirements, Tock currently only supports modern ARM and RISC-V based 32-bit MCUs.

As depicted in Figure 2.7, Tock can be broken down into a set of fundamental building blocks. The `arch` component includes architecture-specific code (e.g., for different Cortex-M families) providing hardware-specific methods to make context switches or kernel system calls [195]. Related, the `chips` component is analogous to a traditional HAL implementation for various MCU specific peripherals (e.g., GPIO, UART, I2C, SPI), in turn, used by the `boards` component to configure a particular Tock platform (i.e., supported development board), defining its proper IO pin states, initializing the kernel, and loading initial applications [195].

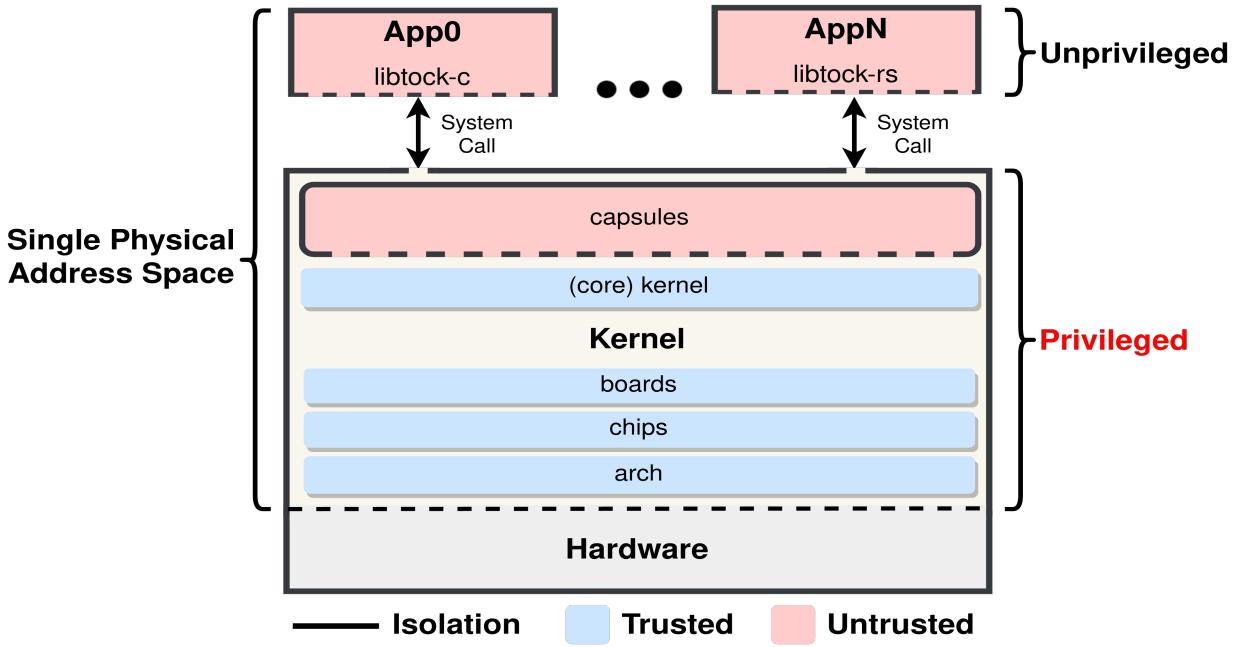

Figure 2.7: Tock OS overview

Tock follows the microkernel approach, in which only essential functionality such as schedulers, system calls, and process/memory management are implemented within a small *trusted* kernel core. Tock `capsules` are additional, *untrusted*, kernel components designed to meet strictly enforced implementation requirements. In particular, each *capsule* is an instance of a Rust struct [104] *sandboxed* by Rust's type-system. Each capsule can directly interact with another through explicitly exposed public fields or functions, but are otherwise governed by Rust's safety guarantees [104]. This allows for fine-grained isolation and interaction between these untrusted capsules, with minimal overhead [104].

Tock establishes a *multiprogramming* environment (Section 4.1) for user application(s), in which each application is compiled and flashed independently from the kernel, then subsequently scheduled (following a round-robin policy by default) to execute as an independent *process*. Processes can be preempted but have a lower priority than kernel events, meaning long-running computations can be executed within processes without worry of blocking [104]. Rather than the Rust-based type system sandboxing used for capsules, processes are confined via *physical memory protection* (PMP), a hardware-based mechanism we discuss further in Section 4.1.1. Use of PMP enables Tock application code to be written in a number of supported languages, however, due to hardware-based isolation, user applications must have an established interface to request kernel services. For this, Tock maintains *userland* libraries for the C [188] and Rust[197] programming languages. Thus, developers primarily write user applications in one of the languages corresponding to said userland libraries. Loaded processes may then interact with the kernel through the set of es-

tablished system calls or with each other using kernel-defined IPC mechanisms [104].

Scheduling within the Tock kernel itself differs from the processes it manages. The kernel is event-driven, meaning kernel execution does not resume until a context switch is triggered by an asynchronous hardware interrupt or system call. Any events that can occur, are statically allocated within an event queue, only made possible because the number of events can be determined at compile-time [104]. When an event is triggered, the kernel (which includes capsules) executes cooperatively and are non-preemptive (i.e., run until completion). However, to avoid blocking for requests requiring physical I/O operations, kernel interfaces (accessible through system calls) process all I/O operations asynchronously, allowing kernel components to return immediately then issue a callback when the offloaded I/O request completes[10] [194].

[10] Although Tock avoids blocking on I/O requests, e.g., through configuration and use of *direct-memory access* (DMA) controllers, any other standard computations that can not be offloaded will block. Consequently, Tock currently does not provide real-time guarantees for interactions with the kernel.

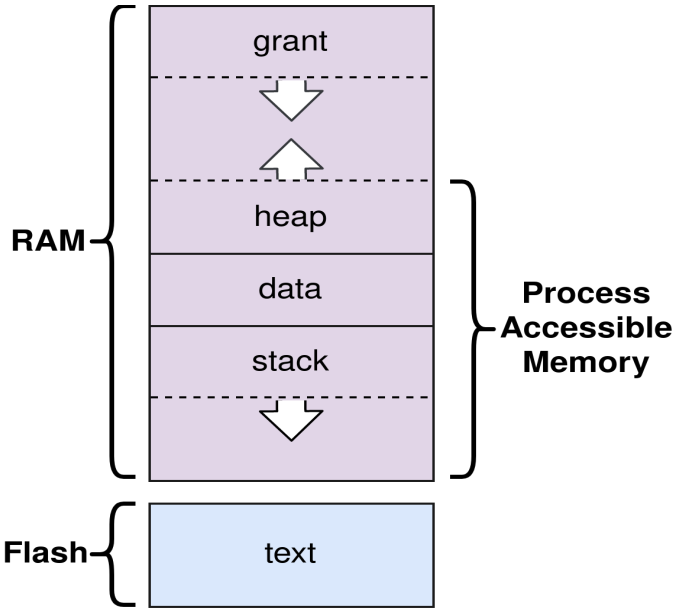

Figure 2.8: Memory layout of a Tock process [190]

Finally, memory management for Tock capsules is handled through a unique abstraction called *grants*. In addition to its program data (i.e., heap, data, and stack regions), each process is provided with a so-called *grant region* at the top of its address space [194]. This region cannot be read from or written to directly by the process (enforced by PMP), rather capsules dynamically allocate (kernel-owned heap) data within the grant regions of processes they interact with [104]. A memory *grant* is allocated when a process first calls a given capsule, leveraging the Rust type-system, references to resources created inside a grant can not escape this region and can be immediately reclaimed if a process dies or is replaced [104].

2.3.4. Further Discussion

Both RIOT and Tock roughly fit the aforementioned general description of an IoT OS, yet evidently differ from one another in underlying design. Overwhelmingly, IoT OSs follow a multi-threading programming model similar to RIOT or a closely related event-driven[11] model [70]. In contrast, Tock deviates from these models by instead providing a multiprogramming-based model. Processes within a multiprogrammed system may be scheduled similarly to tasks within a multi-threaded system, however, as we will discuss, despite the increased complexity of supporting processes, establishing distinctly located and fixed-sized executable applications (e.g., processes) may be necessary when relying on hardware-based foundations (Section 4.1) to provide security guarantees, as Tock does. Isolation enforced using such mechanisms grants Tock with the ability to protect the rest of the system (i.e., single physical address space) from potentially erroneous or attacker-controlled processes, even those written in prevalent programming languages, such as C or C++.

Notably, in addition to hardware-based mechanisms, Tock utilizes software-based mechanisms (Section 4.2) to further reduce its set of trusted components. In computing, *trust* typically implies confidence that a component is guaranteed to always behave in an expected manner for its intended purpose [114]. However, it may also imply a component must be trusted, i.e., there is no other choice, and it is exempted from further checks. The former implies an attacker gaining control of the compo-

[11] In an event-driven system, primary tasks typically only execute once a corresponding external event (e.g., interrupt) is triggered, otherwise execution either remains idle or within a simple main loop.

nent should not be able to make it misbehave (especially in ways effecting the larger system), where in the latter it may be assumed an attacker controlled component may be made to misbehave. Nonetheless, we can see trust is orthogonal to security, thus establishing trust models for all system components (Section 6.2) may help define or even improve the security of an IoT OS. We will examine these ideas further, particularly with respect to the intrinsic design choices forming the foundations of our two selected OSs (Chapter 5).

Although, RIOT shares many similarities with other IoT OSs, it claims to be an OS to rule them all [33], due to its ability to support the whole spectrum of IoT devices. Tock on the other only supports modern 32-bit devices, as they provide the hardware-based mechanisms relied upon by Tock. Thus, RIOT may serve as a prime example of a typical IoT OS, whereas Tock illustrates a novel, but potentially not as broadly applicable, implementation. Despite this, both RIOT and Tock have been deployed in at least one[12] real-life (and seemingly similar) IoT-related scenarios [118, 3]. More generally, because execution begins with the OS and subsequent execution of user applications may be managed by the OS, securing deployments relying on IoT OSs may be best addressed at the OS level. Thus, by researching the security of these two well-established systems, we may achieve generalizable knowledge for the secure development and adoption of IoT OSs.

[12]There may certainly be many more real-life deployments utilizing RIOT or Tock. Notably, many gather at the annual RIOT Summit [159] to discuss the state of RIOT, a potential testament to the traction RIOT has gained in the community.

2.4. Summary

This chapter characterized low-end devices, their relevance and use in the IoT, and how OS support for these devices may be crucial to their development, growth, and, as we will continue to examine, security in the IoT. The rising diverse demands and complexity of these systems coupled with the constraints inherent to low-end devices outline the formidable challenges OS development can and will face, particularly when contrasted with development in the well-established IoC. An interesting conclusion may be that common assumptions held for OSs within the IoC should not be hastily made for IoT OSs, until examined further. In particular, resource constraints on low-end devices impose design differences and challenges for IoT OSs. For example, effective hardware-based security technologies applicable to high-end devices, such as the *trusted platform module* (TPM) [69], become less relevant to low-end devices due to the incurred costs, physical size requirements, and power consumption [1]. Similarly, a well-defined *kernel space* and *user space*, in which the former is presumed isolated and of higher *privilege* than the latter, commonly assumed enforced in IoC OSs certainly may not always hold true for IoT OSs supporting low-end devices. Aspects we explore further with our foray into security, which we begin next by establishing our threat model.

Chapter 3.

Threat Model

Every IoT device follows a life cycle. The start of this life cycle may be traced as far back to initial fabrication of the chip and assembly of the device, then continue through to a software provisioning stage, and finally be transferred to a destination or user for continued use. Without a doubt, potential threats can arise at several points throughout this life cycle. A device–which by our definition, includes the hardware, IoT OS, and the user applications running as part of the OS–will unavoidably be handled by multiple entities, then deployed and used in a wide range of physical environments for varied use cases, some of which can not be anticipated beforehand. Once deployed, these systems can sense and actuate in the physical world or communicate information over some physical or wireless medium. In this chapter, we focus on aspects of the device cycle most relevant to the execution of IoT OSs and the applications they serve. Relevance here is dictated primarily by what any given IoT OS may realistically provide defenses against. Thus, subsequent discussion starts by

identifying entities involved in the IoT device life cycle, allowing us to narrow down those relevant to our discussion, the threats they pose, and high-level goals IoT OSs may seek to start defending against said threats.

3.1. Entities Within the IoT Device Life Cycle

Recalling our prior discussion on trust in computing (Section 2.3.4), we next discuss entities involved in the IoT device life cycle through a similar lens. In this section, we aim to determine the extent of trust inherent to the entities or more importantly, the hardware or software components they manage, develop, or interact with which relate to any given IoT OS. We closely follow and generalize the model proposed by Levy et al. [104] to help identify entities, expanding upon their work where necessary. Entities identified by Levy et al. may be thought of as potentially indistinct or overlapping logical groups comprised of one or more physical entities [109]. These *logical entities* may interact with an IoT device, for a specific reason, at a given point in the life cycle. As we will find, each entity may not be granted with the same level of trust, and this differing allocation of trust is what will form the basis of subsequent discussion on threats posed to and security goals of an IoT OS.

Device integrators. The original logical entities completely in control of integrating MCU hardware and software. In certain scenarios, integrators may design and assemble the printed-circuit board (PCB) themselves through the integration of an MCU and any external electronic components (e.g., network transceivers). The integrator may then provision the MCU with software (e.g., IoT OS kernel). Con-

sequently, the device integrator's liability may be to determine the authenticity of provisioned software (and hardware), as well as modeling against threats expected during the device's life cycle. For our purposes, this provisioning entity must be and is expected to be trusted.

Core kernel developers. Formed by core project members or external contributors involved in developing the core kernel functionality. The core is where code execution begins upon system power on/reset. Thus, unavoidably core functionality developed by these entities must be granted some level of ungoverned trust at runtime, but will ideally be minimal (Section 6.1) in terms of code size and thoroughly audited beforehand (Section 6.9).

Kernel component developers. Formed by core project members, external contributors, or third-party vendors involved with supplying kernel software components eventually directly invokable by others. Software flaws in components such as drivers or system libraries may be mitigated by enforcing additional access restrictions and resource isolation (part of or configured by the trusted core kernel), thus components (and their developers) beyond the core should also be audited and preferable not require trust[1].

Application developers. The entities developing software for a given IoT device to provide end-user functionality. Any developed (user) application(s) loaded onto

[1]Removing trust in certain kernel components is not an easy task. In fact, widely used general-purpose OSs, such as Linux, do not achieve this goal. However, there are exemplary systems, such as seL4 [173], achieving kernel component isolation and consequently gaining the security benefits [40].

the device must adhere to the underlying OSs programming model upon execution. Further, applications may be provided by multiple (potentially distrustful) stakeholders, directly utilize untrusted kernel components, or contain software flaws. Thus, we model applications (and their developers) as untrusted and potentially malicious. Overall, erroneous or malicious access to resources belonging to kernel components and other distinct units of execution developed by the application developers should ideally be mediated and governed by the underlying OS kernel.

End-users. Finally, end-users may arbitrarily interact with a deployed device. This may be for legitimate uses, such as benign interaction or installing/updating applications. However, end-users in particular may also be considered potential attackers (e.g., against publicly accessible devices) and are modeled as such. There should be no trust in interactions carried out by these entities.

3.2. Attacker Capabilities

Given different levels of trust between the aforementioned entities and components, we must consider each introduced component and how they may be interacted with once deployed. Before discussing that which is untrusted, we first discuss that which we trust. Doing so may provide useful context for later discussion relating to our primary focus on attacks originating from untrusted entities and components.

Namely, we define device integrators, core kernel developers, and the hardware and software components they provide as trusted. Consequently, we also consider entities

that may interact with a device before the device integrator as trusted, e.g., those involved in manufacturing the hardware and others within the supply chain. This is not to say security risks may not originate from these entities. On the contrary, attacks at the supply chain level, such as the SolarWinds attack [145], show threats can be introduced even before later entities start interacting with a device. However, we deem such attacks out of scope for this thesis, precluding topics such as secure/verified boot [110, 144]. Instead, we focus our considerations towards threats that may arise after software provided by the core kernel developers[2] is already loaded, i.e., threats that an OS architecture may directly address and threats occurring at runtime.

Shifting our focus to such threats, we defined the following entities as untrusted: 1) kernel component developers, 2) application developers and, 3) end-users. These entities may have malicious intent. Moreover, when considering scenarios demanding multi-tenancy, certain entities may be considered as mutually distrustful stakeholders, for example, competing companies providing separate user applications to be executed by one IoT OS (Section 2.2.1). Such distrust may extend into the kernel itself, where stakeholders may provide kernel drivers or system libraries, granting such components kernel-level access (whether there is such distinction or not). Evidently, user applications and kernel components (both of which are now assumed to be untrusted) may each be thought of as distinct components (Section 2.3). However, an OS's ability to restrict or mediate the execution of each component may govern its

[2]Recall, we defined core kernel developers as trusted, primarily because later entities are reliant upon system properties established by the core kernel.

ability to limit potential damage caused by erroneous or malicious code execution. Any untrusted entity or component has the potential to cause (intentional or unintentional) damage, leading us to model them as an attacker.

We model attacker capability within the context of IoT OSs, focusing on the primary interfaces to the system. Attackers may pose threats locally over physical peripheral-based communication protocols (e.g., UART, USB, CAN, etc), or remotely over RF communications (e.g., USB, Zigbee, etc). We omit attacks conducted over exposed debugging ports, e.g., JTAG or SWD [153], as these give an attacker omniscient view through mechanisms beyond the scope of potential OS-based defenses. We also omit most advanced [36] (potentially remote or physical) passive (e.g., microarchitectural power analysis or timing side-channels [90, 64]) or active (e.g., intrusive fault injection [78] or physical chip decapping [66]) attacks. These omissions may be grouped into potential attacks that may potentially be mitigated by defenses beyond the OS, for example, disablement of debug ports, tamper-resistant casing, or epoxy-based solutions preventing unwarranted access/tampering.

Attackers may carry out various communication or software attacks [201]. We assume attackers are capable of initiating, controlling, sniffing, or modifying communication channels, and can conduct man-in-the-middle (MitM) attacks. However, we assume the Dolev-Yao model [58], where if cryptographic primitives are available and used, the attacker is assumed incapable of breaking these primitives, but may otherwise try and find weaknesses in device security mechanisms or (communication) protocols provided by the OS. Software attacks on the other hand, may target flaws

such as buffer mismanagement [186, 126, 62, 65] or unsanitized input [140, p.268-269], likely exploitable over one or more established communication channel.

3.3. Security Goals

A central problem in OS security is that of resource management [140, p.126][87, p.1]. Ideally, each component should have a restricted/limited set of resources always made available to them, allowing them to solely conduct the tasks required of them and then explicitly and exclusively grant access to specific owned resources to other components, systems, or users. The quintessential resource on a low-end device is memory[3]. More specifically, resources belonging within the single physical address space that code, data, and peripherals can be accessed from. Figure 3.1, inspired by the work of Prinetto and Roascio [147], depicts how an intentional or unintentional operation originating from an external interaction or untrusted system component (outlined in our attacker model) may result in an access violation.

[3]Memory is not the only resource however. For example, we will later discuss that CPU registers may hold critical data or state information as well.

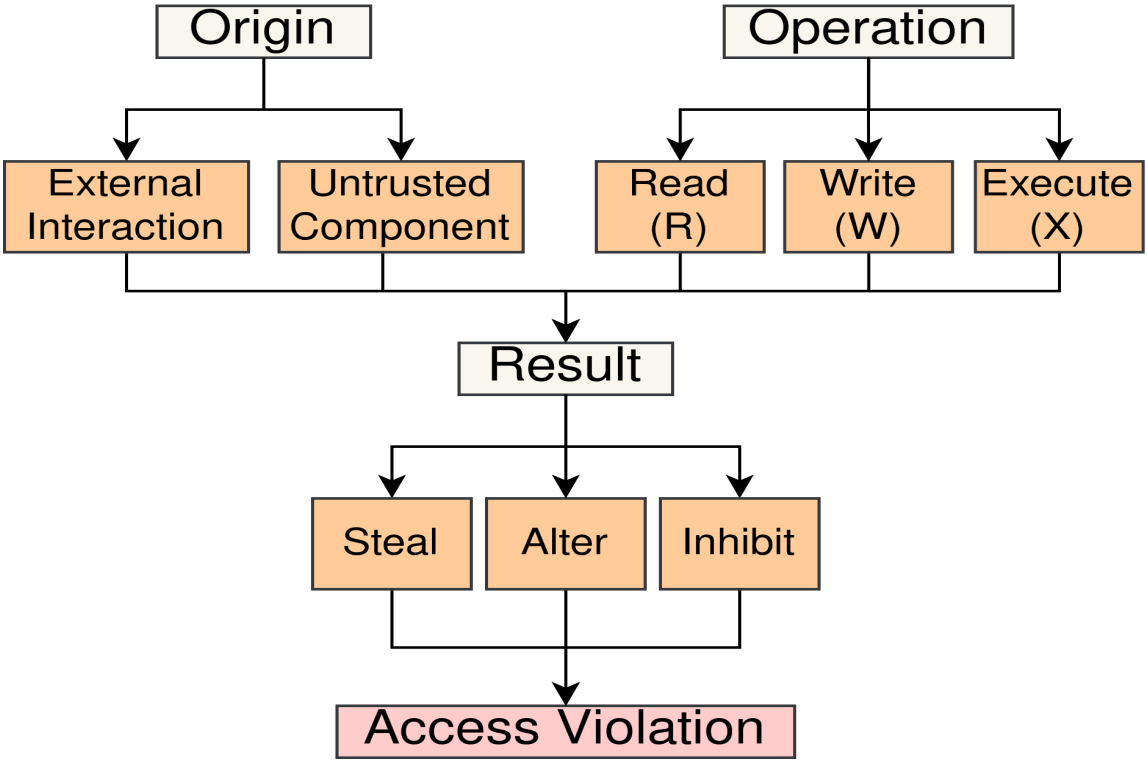

Figure 3.1: Access violation map

Consequently, OS security in general may work towards preventing access violations caused by unwarranted read (R), write (W), and/or execute (X) operations. This may be achieved through establishing, enforcing, and providing security mechanisms to mitigate against the exploitation of system components and limit potential for any access violations by compromised/untrusted components. Naturally, foundational security mechanisms (Chapter 4) start with the trusted core kernel developers, as code they develop begins execution upon system power on/reset before any other. Subsequently executed components (e.g., kernel drivers, system libraries, etc) built atop the core may be subject to mechanisms enforced by the core, but may themselves require non-trivial considerations towards adhering to design principles for

security [140, p.20-24]. General principles or contextualized criteria (Chapter 6) may serve as empirical guidelines[4] to help developers achieve a set of abstract *security goals* to prevent the result of an access violation, such as those comprising the well-established [28, Ch.53][140, p.3] *CIA triad* discussed below:

- **Confidentiality.** Resources belonging to one component are not stolen/read by another, unless given explicit permission to do so. For example, one user application should not be able to arbitrarily read data of another user application or any kernel resource except through well-defined and explicit interfaces.

- **Integrity.** Ensuring any resource belonging to one component or another that is supposed to be in a particular state remains unaltered, except by that which is authorized to do so. Note, beyond cross-component integrity, this may also partially apply to resources within the attacker-controlled component itself. For example, an OS may enforce applications protected by a software-based stack canary [186], where ensuring the integrity of the canary may help prevent against certain stack-based overflow attacks typically used to maliciously alter normal code execution flow.

- **Availability.** Access to information or services provided by certain components for its own or others' use should not be inhibited by an attacker. For example, if a thread/process is modeled as an independent computational resource, any other untrusted component (including any other thread/process) should ideally be prevented from prematurely terminating such computation.

[4] https://www.linzhong.org/opinions/sciencesofsystembuilding.html

Any given IoT OS attempting to provide some level of security may inevitably have to confront the above security goals in an attempt to mitigate against access violations. Yet, involved entities must also confront device constraints, heterogeneity, potentially limited hardware support, and increasingly complex demands such as multi-tenant application support inherent or becoming apparent for low-end devices when designing, building on top of, or deploying hardware utilizing any given IoT OS. Now, with our threat model in place and while keeping these goals in mind, we may start discussing and examining the security of IoT OSs.

Chapter 4.

Security Foundations for OSs Supporting Low-End Devices in the IoT

Foundations are critical mechanisms adopted to uphold certain system properties and goals. Particular to our discussion, security foundations are hardware- or software-based mechanisms typically requiring consideration and tight integration during the initial developmental stages of an OS to effectively uphold its security guarantees. Once integrated, foundations may require significant efforts to change once non-trivially further developed on top of or relied upon. These changes may be required at the kernel or user application level, or both, either of which may alter the mental model and assumptions of developers or users, likely requiring extensive updates to supporting items, such as documentation.

Foundations relate to the security principle of *security by design* [140, p.24] which, in our case, core kernel developers may heed when initially developing their system. In systems security, failing to do so may produce overly vulnerable systems [40], and leave them vulnerable until extraneous incremental efforts towards introducing strong foundations after the fact[1] are made [106]. Correspondingly, security foundations relevant to IoT OSs may help work towards adhering to security by design; any foundation adequately integrated into the system may serve as the critical last line of defense upholding relevant security goals.

If an OS were to become ubiquitous (or advertise such readiness [33]) for low-end devices within the IoT, these foundational security guarantees should be known and understood. Security foundations (or lack thereof) may be critical for an IoT OS to securely satisfy particular use cases, such as those requiring multi-tenancy. In our case, examining foundational hardware- and software-based mechanisms within the IoT OS design space serves as our first step towards understanding the set of high-level criteria such systems may satisfy to define, establish, and maintain security more broadly.

[1] Note, necessary efforts to improve security in the first place might only be realized once the given system has already been identified as lacking strong security guarantees.

4.1. Hardware-Based Security

Hardware-based security [147] involves any mechanism dependent on underlying hardware support to protect certain system resources against access violations made by malicious or untrusted components. In our context, we focus on readily available COTS hardware, as the OSs we reference, whether pertaining to the IoC or IoT, primarily support such hardware. In general, the architectural details of a specific physically implemented COTS CPU are specified by an *instruction set architecture* (ISA). Many ISA specifications are well-established and share high-level commonalities among one another, including those specifying mechanisms relevant to security. However, as we discuss below, no matter how efficient an implementation of an ISA is, certain specifications may better align with specific device classes and their corresponding applications.

To contrast with later, we briefly discuss hardware-based security mechanisms relating to the high-end class of devices. The CPUs on these high-end devices typically implement concepts around *virtual memory* [83], commonly generalized as an embodied unit called the *memory management unit* (MMU), which may be used to restrict accessible regions of memory for any given process. Through the dynamic configuration of an MMU, the OS (kernel) may establish mappings from virtual *pages* (allocated regions of memory) belonging to a process to physical *frames*[2]. As depicted in Figure 4.1, processes are given the illusion of owning their own full pri-

[2]Multiple processes may reference the same physical frame [86]. This is typically achieved through explicit access to shared resources mediated by the OS. However, ensuring safe synchronization between aliased memory is typically left to the application developer.

vate (virtual) address space, but in reality all memory is mediated and mapped by the OS onto the physical address space of the given device.

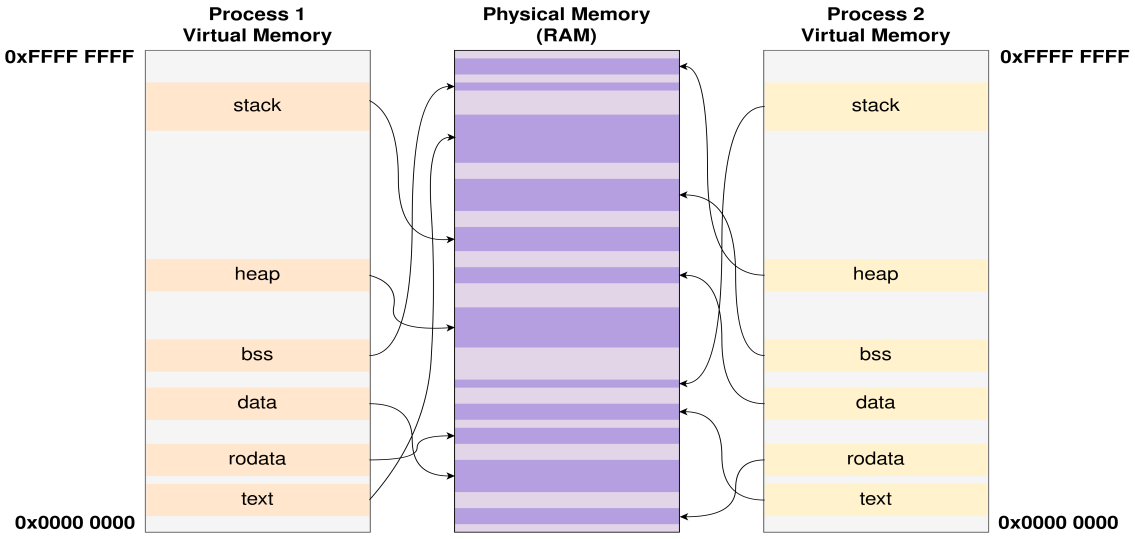

Figure 4.1: Depiction of virtual memory management

If any process could also configure the MMU, all isolation guarantees could be lost. To prevent this, systems rely on multiple hardware-based privilege levels[3] to restrict which instructions can be executed and what registers can be accessed. Most traditional[4] systems run[5] kernel code in Ring 0 (supervisor mode) and restrict user code to Ring 3[6] (user mode). Code in Ring 0 is said to be more *privileged* than code in higher rings and is exclusively granted access to privileged instructions and

[3]Commonly referred to as protection rings [140, p.146-151].

[4]For this particular discussion, we are referring to x86(_64) based systems, however most high-end devices share similar ideas, albeit with different terminology.

[5]Code may be said to be written in a given level and executed/run in a mode [165, p.3]. For example, users may write a device driver in the user level then insert it to be executed in supervisor mode.

[6]Most general-purpose systems today only support Rings 0 and 3, leaving Rings 1 and 2 unused [140, p.150-151].

registers, e.g., those for MMU configuration.

Ostensibly, modern day IoC OSs support *time sharing* through the management and scheduling of multiple processes (independently executable applications), each with their own private virtual address space. For a CPU to directly fetch and execute the instructions corresponding to a given process, each process must be loaded into main memory (RAM) by the OS, as seen back in Figure 4.1.

This scenario resembles that of *multiprogramming* [54, 72]. First termed in the 1950s, multiprogramming permits sharing of device resources among several concurrent processes, each of which occupy a physical region of memory. Multiprogramming brings with it many challenges as each process must be restricted from accessing resources (e.g., memory) of other processes. Failure to ensure protection may lead to erroneous writes or reads from other processes to resources not belonging explicitly to them. Thus, for multiprogramming to be realistically and securely achieved, there must be some mediation and enforced protection of processes by trusted code that is itself isolated and protected from executing processes. Without this, multiprogramming may not be fully realized or may be achieved in a highly unreliable or insecure manner, particularly when faced with malicious or untrusted code.

With privilege modes and an MMU as the foundations, virtual addressing has become widely used and an effective way in actualizing multiprogramming for multi-tenant OSs. In fact, many general-purpose OSs (which overwhelmingly support some aspects of multi-tenancy these days), fundamentally require an MMU [187, p.196], for

example, to support multiple isolated user processes. However, MMU configuration is complex, typically requires non-trivial amounts of cache memory to be reasonably performant [8], and requires more physical circuitry (e.g., transistors) to implement which incur extra costs and take additional physical space, all of which make an MMU less viable for low-end devices. As a consequence, low-end devices (i.e., MCUs) have much simpler memory hierarchies, opting to forgo MMU-based virtual addressing altogether [113, p.20][176] to reduce complexity in memory management and hardware. This raises the question about what hardware-based mechanisms are available to IoT OSs supporting low-end devices to provide software isolation and meet complex demands, such as that for multi-tenancy, in a reliable and secure way; a question we saw being explored through hardware-based alternatives in our brief overview of Tock (Section 2.3.3). Maintaining concepts around multiprogramming as a viable approach to multi-tenancy, we next highlight and discuss this potential hardware-based mechanism and its relation to low-end devices.

4.1.1. Physical Memory Protection and Privilege Modes

Eliminating complex hardware commonly found on high-end devices coincides with the constraints partly defining low-end devices and their applicability to the IoT, yet leaves them lacking the same hardware-based security foundations. For this reason, the ARMv6-M [21] (plus its subsequent versions [19, 20]) and RISC-V [165] ISAs have specified an alternative which may be optionally implemented for the subset of low-end devices based on these architectures, many of which may be found in COTS products [174, 178]. We generically refer to this alternative as *physical memory pro-*

tection (PMP), but note that naming may differ between ISAs [19, p.688][165, p.48]. This section focuses on the high-level conceptual details relating to PMP, rather than ISA specifics.

PMP is a hardware-based security mechanism that may be configured to establish access permissions (read, write, execute) ranging over of a device's single physical address space. Configuring the range spanned by a PMP *region* and its associated permissions typically requires access to a set of corresponding memory-mapped registers. As a consequence, the number of configurable regions is limited due to the finite space available for memory-mapped registers, making PMP an inherently coarse-grained protection mechanism.

Preventing arbitrary access to PMP registers is typically achieved through the introduction of privilege modes to distinguish between *privileged* and *unprivileged* code execution, where access to PMP registers is made exclusive to privileged code. Depending upon configuration, permissions can restrict memory operations made by unprivileged and/or privileged code. Unprivileged code attempting to access memory *outside* of all currently established PMP regions will trigger a system *fault* which, again depending on the configuration, may also apply to privileged access as well. Moreover, any access made *inside* an established PMP region made by privileged or unprivileged code that violates its defined permissions will trigger a fault. In this context, a fault corresponds to an event automatically triggered by hardware resulting in control being forced (a.k.a., *trapped*) to a specific piece of software called

a *fault handler*[7]. Correspondingly, privileged code, such as an OS kernel core, may define the logic within this fault handler, e.g., to notify users or recover from the fault.

Due to the fact that peripherals, RAM, and Flash memory are mapped to the same single physical address space, PMP may be used to restrict access to any of these resources. Evidently, protecting a specific resource will require knowledge of where that resource will/should exist in memory, i.e., its starting address and where it will end (i.e., how large the region should be). In Figure 4.2, we depict one possible configuration scenario relating to multi-tenant OSs supporting multiple independent user applications via a multiprogramming model. Going from left to right, we note that upon system power/reset, the system begins privileged execution at some predefined location, in our example, it is that of an IoT OS kernel placed at the beginning of Flash. The privileged kernel can freely conduct any necessary system initialization then begin scheduling its user applications. Before scheduling the first application (in orange), PMP regions are set over resources established as belonging to the application, this includes its code located in Flash and a predefined size of RAM usable for data storage.

[7]There may be several distinct fault handlers, defined for different exceptions. For example, there typically exists a *hard fault handler* acting as the default handler for any exceptions unhanlded by others.

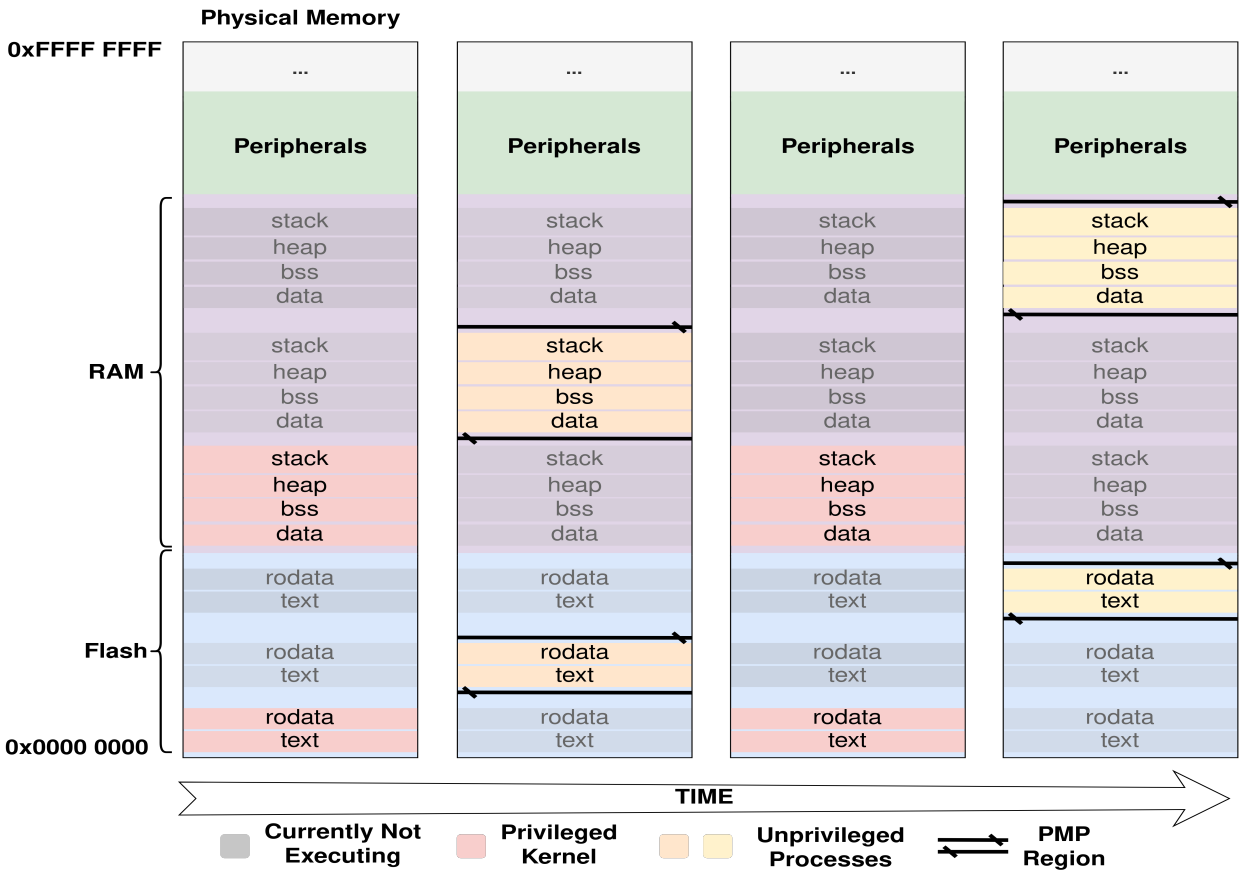

Figure 4.2: Depicting configuration over time for one example of an OS supporting multiple isolated user applications via PMP

Finally, a context switch can occur, in which execution is set to unprivileged and redirected to the applications protected Flash, allowing it to execute. Assuming at some later time an exception/interrupt occurs to grant execution time to the privileged OS scheduler, scheduling another application (in yellow) will require PMP be similarly (re)configured[8]. In both cases, the currently executing application will be *confined* to the established PMP regions and subject to the permissions of said

[8]Given the kernel is privileged (and assumed trusted), it may disable PMP temporarily to allow itself to reconfigure it, then enable it again before scheduling execution of any untrusted code.

regions. Any access violation will result in execution trapping to a kernel-owned fault handler. Overall, this brief outline and contrived example shows how, in theory, dynamically configurable physical memory protection and privilege modes may serve as security foundations for multi-tenant IoT OS supporting low-end devices.

4.1.2. Case Study: ARM Memory Protection Unit

The ARM *Memory Protection Unit* (MPU) is a real-world instance of what we have been generically referring to as PMP. The operational details of the slightly varying MPU versions are specified under each updated version of ARM's *Protected Memory System Architecture* (PMSA), the most recent of which is part of the ARMv8-M ISA [20]. These specifications are made available as an optional feature[9] in the implementation of an ARM processor, such as those in the Cortex-M3 [18] and Cortex-M4 [22] family of processors[10]. Later in this thesis, we conduct experiments atop a device (Figure 5.1) containing an MPU conforming to the ARMv7-M ISA [19], thus the details in this section largely follow this specification. However, we summarize relevant changes [27] made by the ARMv8-M specification at the end of our discussion.

[9] Meaning those implementing the ISA (to build a physical processor) may choose to include or exclude it, remaining compliant with the ISA in either case.

[10] These processors then become the core of an MCU central to our definition of a low-end device.

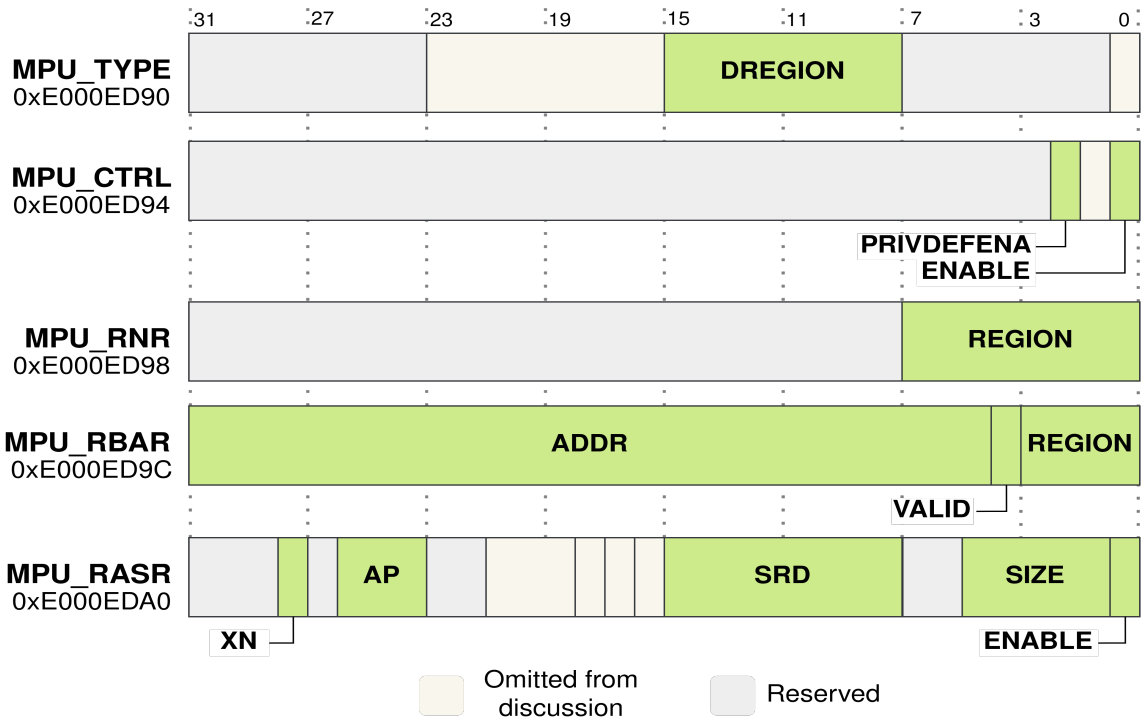

Figure 4.3: ARMv7-M MPU registers

In Figure 4.3, we outline the primary registers of interest when configuring the MPU on an ARMv7-M based device. The name of each register and its corresponding memory-mapped address are given on the left. Note, each register is mapped within a region reserved as a *Private Peripheral Bus* (PPB), an area of memory holding critical resources such as these registers. The PPB can never be executed, referred to as being *Execute Never* (XN) in the specification[19, p.648], and only accessed in the privileged mode [19, p.744].

We will be focusing on the fields highlighted in green for each respective register. As denoted, we omit details of some fields not wholly relevant to later experimenta-

tion. Those marked as reserved are not in use, as specified by the ISA, reading from these fields should yield 0's and writes will be ignored.

To start, determining the number of configurable regions supported by a device's MPU is done by reading the `DREGION` field within `MPU_TYPE`. For ARMv7-M, this will either be 0 (MPU not supported) or 8 (MPU available), making 8 the maximum (and only) number of supportable regions. The MPU can be enabled by setting (i.e., writing a value of 1 to) the `ENABLE` bit within `MPU_CTRL`. By default, any access made by both privileged and unprivileged code not coinciding within any configured MPU region will result in a system fault. However, if the `PRIVDEFENA` bit in `MPU_CTRL` is set, the default memory map [19, p.648-649] (what would normally be accessible) is applied for privileged access made outside any MPU region. Any unprivileged access attempts will remain confined to regions explicitly configured as accessible.

To select a region to configure, the region number (between 0 and 8 in our case) to configure can be written to `REGION` in `MPU_RNR`. Now, subsequent configuration written to `MPU_RBAR` or `MPU_RASR` will apply to this region. In ARMv7-M, regions may overlap, where any access made to an address with overlapping regions becomes subject to the permissions of the highest numbered region.

The `MPU_RBAR` register is used to set the base (i.e., starting) address of the currently selected region. Alternatively, when writing to `MPU_RBAR`, the base address of any other region may be configured by setting the `VALID` bit and region-to-be-configured in the `REGION` field of `MPU_RBAR`. This allows setting the base address of any given

region (including one not currently selected in `MPU_RNR`) in a single write, without having to write to `MPU_RNR` first. In either case, the value written to `MPU_RBAR`'s `ADDR` field will determine the region's base address, where any base address is required to be a multiple of the selected region's size. For example, if a region has a size of 16 KB, the base address may be $0x4000 * n$, where n is an integer resulting in an address within the device's memory space.

`MPU_RASR` is used to configure the size, access permissions, and state of the current region selected in `MPU_RNR`. The size of the region (starting from its base address) is $2^{(SIZE+1)}$, where `SIZE` is set by the correspondingly named field in `MPU_RASR`. However, regions must be at least 32 bytes in size and a power of two. These limitations mean creating regions of arbitrary size may take several regions of smaller size. To provide some flexibility, any region greater than 256 bytes is subdivided into eight equally sized subregions (each having the same permissions). Each subregion is enabled by default, but may be selectively disabled by writing to the `SRD` field. Next, the three bits spanning the `AP` field are used to set read and/or write permissions for privileged and/or unprivileged access. Any privilege mode given read access may also execute instructions, unless the `XN` bit is set for the given region. Finally, setting the `ENABLE` bit ensures the region is enabled when the MPU is itself enabled.

Given the ARMv7-M MPU specification, we learned of limitations regarding the size of a given region, such that certain scenarios may need to utilize several (potentially overlapping) regions to realize a larger region of an intended arbitrary size. This is particularly problematic [221] given the limited number of configurable re-

gions (8 in ARMv7-M). To address this, the more recent ARMv8-M ISA [20] provides several notable improvements. First, MPU regions are allowed to be any arbitrary size that is a multiple of 32 bytes. Further, the number of regions that may be implemented/provided has been increased to 16. These improvements may provide more flexibility and granularity, but will only be found on more recent devices based on the ARMv8-M ISA. Overall, any OS broadly supporting ARM-based devices and utilizing an available MPU will be required to consider the differences between ISA versions and inherent limitations with respect to region granularity.

4.1.3. Further Discussion

Our focus on PMP and privilege modes does not encompass every possible security foundation available, even among currently available COTS hardware. However, these foundations have served as the most widely utilized hardware-based security mechanisms among currently developed IoT OSs supporting low-end devices and are indeed the only seen foundations supported by both OSs covered in this thesis. Referring back to one of these systems, Tock, we note that in addition to those based upon ARM, Tock supports devices based upon RISC-V, relying on similar security foundations specified by the RISC-V ISA.

RISC-V is a more recent free and open ISA first publicly published in 2011 [163] which has seen active development and increasing adoption, particularly for use in low-end devices. The RISC-V ISA is extensible. Processors implement a base set of unprivileged instructions then may optionally provide several extensions specified

by unprivileged [164] or privileged [165] ISAs. Notably, the privileged ISA specifies several privilege modes [165, p.3] and PMP [165, p.48-53], both of which may be optionally implemented and included for use in low-end devices. RISC-V PMP and privilege modes closely relate to our prior discussions, thus serve as viable instances of hardware-based security foundations for IoT OSs.

Moving beyond PMP, we briefly mention an additional security extension first specified within the ARMv8-M ISA, typically referred to as TrustZone-M [20, p.180-183]. This feature is not utilized by either IoT OS we cover and only supported by a small subset of ARM-based devices [25]. Regardless, low-end device's supporting TrustZone-M have the option to split execution into two distinct *security levels* (a.k.a., *worlds* [111]): the *Secure World* (SW) and the *Non-Secure World* (NSW). Additionally, the extension defines three memory *security attributes*: *non-secure* (NS), *non-secure callable* (NSC), and *secure* (S). Each world acts orthogonal to privilege modes and may be assigned non-overlapping (Flash, RAM, and/or peripheral) memory regions marked with security attributes. The SW can access all memory regions while the NSW can only access regions marked as NS or NSC. Confined to NS regions, NSC regions act as small entry points that the NSW may use to request services from the SW. In summary, TrustZone-M is a lightweight mechanism to further partition a system, providing the option for lightweight secure function calls.

TrustZone-M exemplifies how hardware-based security mechanisms are always evolving and trying to mitigate against threats, which may be of particular importance when regarding growing fields such as the IoT. Yet, introduction of hardware-

based mechanisms, in general, makes hardware and software design more complex and may still leave devices vulnerable to attack [111, 221, 130] (Chapter 5). This does not diminish their importance but alludes to the fact that system designers must take into account hardware and software when designing secure systems.

4.2. Software-Based Security

The need for hardware-based mechanisms in the first place primarily arises from one of the most notorious challenges facing OS security: software vulnerabilities. Software-based security foundations work towards preventing these vulnerabilities. More specifically, the existence of developer-written software containing *spatial*, *temporal*, or *concurrency* errors inevitably leading to *memory corruption* [186]. Spatial errors may occur via integer, stack, or heap based overflows/underflows, e.g., leading to out-of-bounds reads or writes. Temporal errors may occur via mistaken access to memory pointers no longer pointing to valid resources, e.g., extremely subtle use-after-free bugs resulting in *dangling pointers* pointing to no longer valid dynamically-allocated resources [93]. Moreover, the OSs we have reviewed thus far run multiple concurrent units of execution that interact with one another and underlying kernel components. Handling multiple *aliases* (references) to shared resources is thus crucial to prevent concurrency errors [93]. In general, any aforementioned error may, at the very least, lead to unexpected behavior or system faults but may also lead to attacks allowing for the corruption of data, leakage of sensitive information, or alteration and control of normal execution flow.

Consequently, attacks successfully exploiting an OS kernel may compromise or circumvent an OSs ability to further mediate resource access. In particular, arbitrary or unwarranted access to memory may allow for [51] violation of established isolation boundaries between user processes, complete control over device drivers, or provide direct access to any underlying hardware made available through MMIO [28, Ch.36] interfaces.

In general, and due to lack of prevalent alternatives, a vast majority of OSs are written in C or C++, including those targeting low-end devices [138]. These *low-level* systems programming languages [51] provide flexibility and performance through minimal (or non-existent) runtime environments, making them suitable for OS development. Differing from those providing a managed runtime (e.g., garbage collection), low-level languages traditionally require the programmer to manually manage, allocate, and free resources, e.g., pointers to memory and dynamically allocated heap memory. Rather infamously [186], these languages lack compile-time safety guarantees making it easy for developers to make programming errors, potentially leading to security vulnerabilities or unexpected/undefined behavior. This incurs a huge burden on fallible developers, requiring an immense amount of expert knowledge, caution, and time to develop resulting error-free code [121].

4.2.1. Memory-Safe Programming Languages

In contrast to programming languages with weak memory safety guarantees, *memory-safe* languages make it so developers must instead go out of their way to introduce

memory-related errors, instead of perpetually mitigating against them. Programming languages serve as a foundation for the development of an OS, enabling developers to reason (with varying degrees of clarity) about the semantic correctness of their programs in the face of memory corruption errors. For a systems language to achieve this, it may enforce a strong *type system* and provide additional constructs to help programmers write memory-safe code [93].

In addition to programming languages, there are methods for static or dynamic analysis which examine a program before execution (i.e., its source code) or during execution respectively. Static and dynamic analysis may work towards detecting or reducing the amount of memory corruption errors and may be integrated at anytime during the development of an OS. Despite years of rigorous testing through static analysis and automated testing, well-known general-purpose OSs still struggle with memory related vulnerabilities [121, 106]. This does not mean they should not be considered in the overall developmental life cycle of an IoT OS (Section 6.9), but raises the question on whether memory-safe systems programming languages may be an effective and viable alternative, particularly for new systems or those in early development. If providing strong guarantees upon correct utilization, a memory-safe language may help satisfy the principle of security by design and serve as a foundational aspect for the development of secure (IoT) OSs. We next discuss Rust [117], a rising systems programming language offering strong memory safety guarantees and novel abstractions relevant to OS development.

4.2.2. Case Study: Rust

Rust is a *type-safe* systems programming language designed to largely eliminate undefined behavior, provide memory safety, and minimize the amount of memory unsafe code. Type-safety refers to Rust's strict type system (e.g., the defining characteristics developers must adhere to when writing software) enforced at compile-time to prevent a majority of spatial, temporal, and concurrency errors by eliminating buffer overflows, accesses to uninitialized or deallocated memory, data races, and more [117, 92]. Notably, Rust does not require garbage collection and supports manual memory management, making it a good target for systems programming.

Memory safety is primarily achieved in Rust through the concepts of *ownership*, *borrowing*, and *lifetimes*. Additionally, *traits* provide useful abstractions over a collection of methods that certain types must implement to satisfy the given trait. Finally, the `no_std` environment and `unsafe` keyword are essential for low-level programming, e.g., developing OSs. We briefly outline these concepts below with short examples for later context.

Ownership. In Rust, objects always have a unique owner [93, 50]. Variables bound to an object are granted ownership over said object. If ownership is released, i.e., the object goes out of scope, memory is automatically deallocated [129, 34] for that object. This allows Rust to place restrictions on mutable *aliases* (references) to a given object [93] and stop issues pertaining to aliasing at compile-time [50].

```rust
fn foo(a: String) {
  // drop(a);  // Inserted automatically by the compiler.
}

fn main() {
  let s = String::from("Hi");  // Create owned string
  foo(s);  // Transfer ownership of s into foo()
  println!("{}", s);  // ERROR: borrow of moved value
}
```

Listing 4.1: Rust ownership example, derived from [132, p.10]

Shown above, passing `s` by value into `foo()` transfers ownership of `s` to `foo()`, making it unavailable for subsequent use, particularly because `s` may get deallocated once `foo()` terminates. Note, calling `drop()` in `foo()` to deallocate the owned value is optional and would occur automatically regardless, hence why we leave it commented out.

Borrowing. Intertwined with the concept of ownership, borrowing allows temporary references to owned values. Additionally, to avoid dangerous errors [140, p.273] Rust requires users explicitly define whether their value is mutable, defaulting to immutable if not specified. Thus, `s` in the below example is guaranteed immutable throughout the programs entire execution.

```rust
fn increment(i: &mut i32) {  // Borrows mutable reference
  *i += 1;  // Dereferences then increment
}

fn main() {
  let s = String::from("Hi");  // Create owned string

  let mut i = 0; // Explicit mutable declaration

  bar(&s);  // Borrow s for duration of bar()
  println!("{}", s);  // Print owned value

  increment(&mut i);  // Borrow mutable reference to i
  println!("{}", i);  // Print incremented value
}
```

Listing 4.2: Rust borrowing example, derived from [132, p.11]

In turn, Rust enforces that any value can only have one mutable reference or many immutable references [50] at a given time, allowing the compiler to catch unwarranted or unsynchronized data access attempts, evidently the cause of many systems programming errors [92]. However, as we will see next, there are well-defined limitations on borrowing values, thus, a supporting concept is needed to help Rust manage references.

Lifetimes. Due to the possibility of resource sharing to occur between different scopes defined within a program, Rust defines lifetimes to manage and stay informed on how long any given resource will exist. In the example below, a lifetime of $'a$ (lifetimes start with the $'$ character and can have arbitrary names, with a few exceptions) is given to a resource within a *struct*. Following in `main()`, the variable `x` defined in the outer scope is set to reference `t.val` with a shorter scoped lifetime,

leading to compile-time errors. Overall, lifetimes mitigate against large swaths of memory corruption errors, such as subtle use-after-free bugs.

```
struct Baz<'a> {
  val: &'a i32,
}

fn main() {
  let x;  // x lives as long as main()

  {  // Start of temporary scope
    let t = Baz { val: &5 };
    x = &t.val; // ERROR: `t.val` does not live long enough
  }  // ERROR: `t.val` dropped here while still borrowed

  println!("{}", x); // ERROR: borrow later used here
}
```

Listing 4.3: Rust lifetimes example, derived from [132, p.11]

Evidently, annotating lifetimes is a tedious task, luckily, the Rust compiler can infer lifetimes in a majority of use cases. Thus, this feature allows Rust to automatically manage resources efficiently, only requiring explicit user annotations for advanced use cases.

Traits. An abstract declaration defining a set of methods, all of which must be implemented by any type wanting to satisfy a given trait. These may provide abstractions over interfaces to enforce expected input and resulting output. Not wholly unique to Rust, and similar to polymorphic interfaces in other languages [42], traits may be used to define specific behavior [129].

```
1  struct Rectangle { l: f64, w: f64, }
2
3  pub trait Shape {  // Any impl of Shape must define area()
4    fn area(&self) -> f64;
5  }
6
7  impl Shape for Rectangle {
8    fn area(&self) -> f64 { self.l * self.w }
9  }
10
11 fn main() {
12   let s = Rectangle { l: 2.0, w: 3.0 };
13   println!("{}", s.area());
14 }
```

Listing 4.4: Rust traits example

Interesting to note, the `pub` access modifier for `Shape` allows other Rust code to import and utilize/implement the corresponding trait methods. Thus, omitting these identifiers may allow for fine-grained and differing access controls to the set of methods available from a given type implementing certain traits.

The `no_std` environment. Rust provides a rich standard library [171] for the development of portable applications among OSs, providing additional abstractions and generally useful types. However, for those targeting environments not supported by a common OS, such as bare-metal embedded systems and OS development (as is our case), there is no underlying system the standard library may interface with. Thus, bare-metal applications may only rely on Rust's core library [170] `libcore`. To denote sole dependence on `libcore`, Rust libraries (a.k.a., *crates*[11]) specify the

[11] Libraries are commonly managed by the *cargo* project management tool, and are thus aptly named crates.

`#![no_std]` *attribute* at the top of their source. Excluding Rust's standard library does not remove any concepts discussed thus far and a `no_std` environment is assumed for any related discussion hereinafter.

The `unsafe` keyword. Most importantly, not all operations in Rust (or any low-level language) can be guaranteed to be completely safe. Fundamentally, any OS must have direct memory access to certain underlying resources. In particular, direct physical memory access, whether it be to typical memory or MMIO, is inherently volatile and may be changed by other software components, the hardware itself, or through environmental interaction. In traditional memory unsafe languages, such as C, no notice is taken and direct access can be achieved by dereferencing raw pointers to memory. However, in Rust, such arbitrary dereferencing violates compile-time memory safety guarantees [99]. Thus, to allow necessary interactions, Rust defines a property called *unsafe* which instructs the compiler to bypass safety checks on components explicitly marked with the `unsafe` keyword. Unsafe code is then granted the ability to conduct operations that may result in undefined behavior, such dereferencing raw pointers, modifying mutable static variables, and calling other unsafe code [99].

```
let addr: usize = 0x1234;
let p = addr as *mut usize;

unsafe { // DENIED
  let x = *p; // Requires unsafe
}

unsafe { // DENIED
  *p = 123; // Requires unsafe
}

let p: fn() = unsafe { // DENIED
  core::mem::transmute(address as *const ()) // Requires unsafe
};
p(); // Not reachable without unsafe keyword

unsafe { // DENIED
  asm!("mov r1, r1"); // Requires unsafe
}
```

Listing 4.5: Several lines of Rust code requiring the `unsafe` keyword

Unavoidably, `unsafe` may produce security critical errors due to lack of compile-time safety guarantees [34]. However, because all unsafe code is required to be explicitly marked with the `unsafe` keyword, these sections immediately standout and may ease in future code audits. Related, if a memory safety error does occur, it may be assumed the problem was within defined `unsafe` sections of code. Overall, this provides an explicit interface for developers to carry out necessary low-level tasks, potentially making the cost of testing and validating Rust code lower than traditional unsafe systems languages [29].

4.2.3. Further Discussion

Widely adopted IoC OSs for high-end devices, such as Linux, now contain tens of millions of lines of code and are continuously growing. Such systems lack things such as language-based memory safety in the kernel and as a consequence security vulnerabilities are constantly introduced and discovered, despite processes for static analysis and code review. Attempts at introducing memory-safe alternatives may now only be done so incrementally [106]. In the IoT however, there has not yet been an overwhelmingly widely adopted OS supporting low-end devices. Utilizing software-based security foundations in new systems or considering them for use in yet to be broadly adopted systems (i.e., those more malleable to change) may be more viable in the IoT.

Indeed, Rust in particular has become of increasing interest to researchers and system designers alike. In addition to Tock, there exists several OSs and lightweight runtimes targeting low-end devices relying on Rust to provide safe concurrency and encapsulate required use of unsafe code [202, 216, 39, 168]. Moreover, several C-based OSs supporting low-end devices, including RIOT [161], are exploring how they may integrate Rust for the development of their systems [107, 71]. Concurrently, research has also begun into novel general-purpose OS designs relying purely on Rust's memory safety guarantees to uphold important security properties, such as isolation, fine-grained access control, and fault isolation/recovery [129, 42].

However, a looming threat inherent to all these efforts remains: unsafe code.

Proper encapsulation of unsafe code, audits of said code, and prevention of unsafe usage when found unnecessary may all help reduce this threat. Interestingly, research into how encapsulated unsafe Rust code may be formally verified for it correctness [91, 92, 34] may become an important next step in reducing potential errors from arising when relying on Rust for memory safety and security.

Overall, memory-safe programming languages, such as Rust, may be an important foundation in the development of any system, including IoT OSs supporting low-end devices. Software-based security mechanisms in general are of particular importance to supporting low-end devices as a subset of these devices may lack alternative hardware-based security foundations, such as those relating to PMP. However, an uncaught software error, e.g., within unsafe Rust code [181], could leave an entire system vulnerable, especially if software-based mechanisms are solely used to uphold security. Thus, depending on availability and viability, consideration of hardware-based foundations for use in tandem with software-based foundations may be important.

Chapter 5.

Examining Security Foundations through Experimentation

The inherent constraints introduced by low-end devices pronounces the fact that IoT OSs supporting such devices lack the same hardware-based security foundations commonly utilized within the IoC, and instead must rely on alternative mechanisms or foundations for security. Moreover, due to their heterogeneous nature, some low-end devices may not have any hardware-based alternatives at all, leaving purely software-based foundations as the only viable option for bettering security. Regardless of the approach taken however, the dominant resource on low-end devices remains the same: memory, i.e., the single physical address space shared by all running software in which peripherals, RAM, and Flash are mapped to and accessed from (Section 2.1.1). Thus, preventing unwarranted access violation to memory is central to both hardware- and software-based foundations.

Security foundations however, were said to be an important *initial* design decision, as later integration may either prove difficult or require extraneous efforts of incremental integration. In this chapter, we take a closer look at the utilization of security foundations through a set of hands-on experiments. These experiments aim to examine how actively developed IoT OSs with academic origins, RIOT and Tock, utilize and employ security foundations. Experiments aim to identify if and how foundations are used, determine any additional foundations or mechanisms, and highlight whether the utilization of security foundations from the start has importance to the development of secure systems. Further, experiments may allow us to relate security foundations to identified trends found relevant to low-end IoT, such as the relevance of multi-tenant IoT OSs.

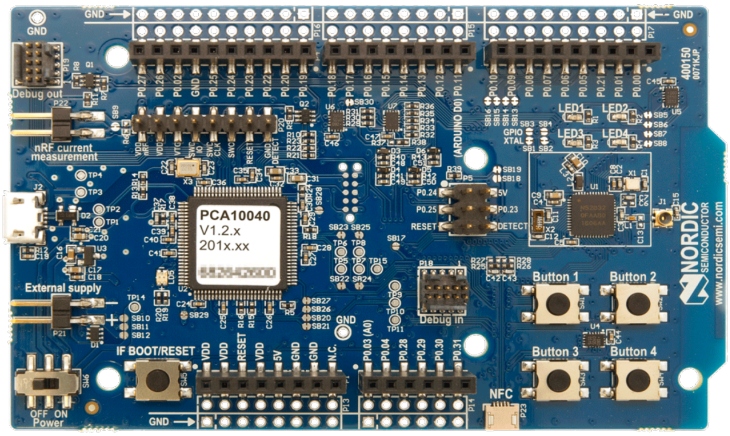

Figure 5.1: ARM-based nRF52-DK development board

The experiments presented in subsequent sections are conducted on the device de-

picted in Figure 5.1, an nRF52-DK development board (device) [174] based upon an ARM Cortex-M4 [22] MCU [135]. This device is supported by RIOT and Tock and contains an MPU adhering to the ARMv7-M ISA (Section 4.1.2). To gain insight into program execution flow and memory accesses during execution, we rely heavily upon physical access to our test device's debugger port and debugging software[1] to interact with said port. However, such access is purely for examination purposes and for insights into our target device's state, including how it may change at any point in time during normal execution flow or right after given external input. To model any external input in a consistent and straightforward manner, a subset of our experiments involve communicating with our device via it's *Universal Serial Bus* (USB) interface, an interface fully supported in software by RIOT and Tock. We note the nRF52-DK does contain wireless BLE communication capabilities, meaning each experiment may theoretically be similarly reproduced using wireless communications to better resemble a real-world IoT deployment. However, such communication depends on software support provided across different IoT OSs, which, in our case, is not adequately consistent between RIOT and Tock (Section 6.7). Modeling interactions over USB avoids this inconsistency and does not produce observations that could not be replicated over other communication protocols/mediums.

In general, determining the effectiveness of mechanisms underpinning a given system through experimental validation, whether it be relating to security or otherwise, holds considerable importance to systems research [139]. In fact, the importance of experimentation has led to the development of many standardized frameworks for

[1] https://www.gnu.org/software/gdb/

examining different aspects of a system. For example, there exists a variety of attack frameworks [208, 220, 212] providing standard methods for testing the coverage of defense mechanisms on Linux-based systems. Notably, such frameworks and related systems research predominantly focuses on systems such as Linux, likely due to their prevalence and standardized nature. As a consequence, many frameworks found in the literature today have built-in assumptions towards a common set of underlying interfaces provided by systems they will be run on. Assumptions, such as a standard set of system calls and shell environment, allow for meaningful comparison between similar systems with differing defense mechanisms. These assumptions however make such frameworks less relevant (or completely unusable) on systems deviating from the norm. For example, there is currently no de facto standard set of interfaces which IoT OSs supporting low-end devices commonly share, this is unlike what has taken place between UNIX and Linux based-systems found in the IoC. Thus, in this thesis we develop experiments ourselves to focus on memory access violations and the role of foundations in preventing such violations.

Interestingly, Mullen and Meany [126] conduct a set of experiments on FreeRTOS [6], a lightweight RTOS not originating in academia but widely used in industry. They identify and assess mechanisms used to prevent two attacks against FreeRTOS applications vulnerable to *stack-based buffer-overflows* (Section 5.3). Their first attack tries to alter normal execution flow and the second tries to inject and execute arbitrary code. As part of our own assessment, we independently recreate attacks similar to theirs in Section 5.3 and Section 5.4 respectively, both for RIOT and Tock. Additionally and prior to these two recreated experiments, we conduct another two

experiments to broadly assess access capabilities of user applications (Section 5.1) and kernel components (Section 5.2). Finally, we conclude with an additional experiment (Section 5.5) involving an attempt to maneuver around defense mechanisms employed by RIOT and Tock.

5.1. Application Access (E1)

Research has made it evident that software running on low-end devices may be exploited just as software on any other device [167, 143, 65, 62, 126]. We also know that the result of such exploits may lead to an access violation from one software component to another part of the system (i.e., memory location). However, in addition to the exploitation of vulnerable components, our threat model further includes potentially untrusted components, in which certain components don't necessarily have to be exploited to be considered malicious, leading them to be of particular concern with respect to system security. Untrusted components highlight potential scenarios of multi-tenancy within the IoT, scenarios in which IoT OSs may try to support.

This experiment aims to generally assess what system resources (i.e., memory) user applications (trusted or not) may access and whether each OS governs such access. The goal is to identify security mechanisms employed to mediate or confine user applications.

Background. Ostensibly, modern IoC OSs have security mechanisms in place for mediating/preventing access attempts coming from untrusted components (particu-

larly user applications) to certain resources, such as systems calls for accessing kernel resources and enforcing access controls over shared file systems. However, an underlying assumption in the development of these mechanisms is that components can not directly access the resource in question in the first place. In the IoC, this assumption is ubiquitously granted through foundations such as privilege modes and a MMU. Without them, there would be no distinction between unprivileged and privileged, resulting in little difference between user applications and kernel code. Further, without private virtual address spaces user applications could directly access any physical memory location, defeating the purpose of things such as system calls controlling file access.

Focusing back on low-end IoT devices and recalling Figure 2.2, the relevant types of memory mapped within a single physical address space are: peripherals (MMIO), RAM, and Flash. Note, this entire single physical address space is occupied and shared by the kernel and user applications, but access to this shared space may be restricted. Thus, for each OS, we will discuss whether access attempts made from a user application to each type of memory is direct, indirect (i.e., mediated), or denied (i.e., confined).

On our ARM-based test device, peripheral, RAM, and Flash memory exist within each respective address range outlined in Figure 2.2. Physically however, our device only spans a subset of this available range with its provided 512 KiB of Flash and 64 KiB of RAM. Thus, memory accesses will always be between the starting addresses given in Figure 2.2 to the end of physically available memory. Accessing arbitrary

addresses within this range can be achieved through the use of pointers and/or *inline assembly* code, as illustrated in Listing 5.1.

```
1  uint32_t addr = 0x20000000;
2
3  *(uint32_t *)addr = 0x46f7;                    // WRITE
4  __asm__("blx %0" : : "r"(addr | 1));           // EXECUTE
5  // Or execute via: ((void (*)(void))addr)();
6  printf("%lx\n", *(uint32_t *)addr);            // READ
```

Listing 5.1: Pointer manipulation in the C programming language

One exception to rule of direct memory access arises when trying to alter Flash memory. Due to inherent electrical properties of Flash memory limiting write/erase operation speeds [75, p.22], directly altering Flash memory is typically disabled [135, p.29-30] or locked [23, p.62-63] by default. Instead, altering Flash memory (which is organized as fixed sized chunks called *pages*) is typically done through a memory controller peripheral [135, Ch.11] made accessible through MMIO. In summary, altering Flash memory is typically indirectly achieved by writing to a specific peripheral, but reading is done directly in the same way as any other memory access. With that in mind, we may now explore using pointers to read, write, and execute memory of our choosing to assess whether each operation is successful or intentionally denied by certain security mechanisms.

Setup. This experiment involves assessing the extent to which users applications running atop RIOT and Tock may access each memory type (peripheral, RAM, and Flash). Through several distinct user applications, we conduct a *memory walk* over the range of addresses pertaining to each respective memory type. Each walk simply

consists of a code loop testing whether each memory operation (read, write, execute) successfully completes on addresses within respective memory ranges. Through a combination of print statements and debugger introspection we observe and confirm each operation. Any unsuccessful operation will typically culminate in an observable CPU fault. Thus, when operation(s) are unsuccessful, we identify why and continue to assess the range of access *not* resulting in a fault (i.e., what the application *is* allowed to access).

Assessing E1 on RIOT. In RIOT, user and OS code are directly compiled together as one monolithic binary. As a result, there is little distinguishing the difference between memory belonging solely to the user application as opposed to the kernel. This means, for example, OS interfaces such as system libraries or device drivers are directly invoked by the user application (rather than through some form of indirection, such as a system call).

Peripheral Access. Invoking an OS interface is later seen in Listing 5.11, where lines 8 and 9 are functions provided by the OS to access MMIO. Taking the `gpio_clear()` function on line 8 as an example, we can see its underlying implementation (Listing 5.2) is actually a direct memory dereference (to clear a bit within the memory-mapped GPIO port). This alludes to the fact that user application code directly accesses peripheral memory in such a way that can be made independent of any OS interface.

```
1 void gpio_clear(gpio_t pin)
2 {
3   // Writes to address 0x5000050C via raw pointer dereference
4   port(pin)->OUTCLR = (1 << pin_num(pin));
5 }
```

Listing 5.2: RIOT `gpio_clear()` implementation

Through code given in Listing 5.3, we generalize direct peripheral memory access allowed in RIOT user application code. We define three memory ranges via the `ranges` array, mapping out the greater part of all accessible peripherals on our selected platform [135, p.24-25]. The first two ranges (lines 2 and 3) contain the MMIO peripherals used by our device to communicate and interact with the physical world; the last range (line 4) pertains to our device's PPB, which we recall holds critical registers, such as those used to configure the MPU, only accessible in the privileged mode. The code below iterates all three of these ranges, conducting a read and write operation at several points within each range. For completeness, we selectively tested execution of peripheral addresses using method(s) outlined Listing 5.1. However, all device peripheral memory is set as non-executable by default [19, p.648] on our ARM-based test device (regardless of whether the MPU is enabled), which we confirm to be the case.

```
1  uint32_t ranges[][2] = {
2      {0x40000000, 0x40026000}, // APB peripherals range
3      {0x50000000, 0x5000077c}, // AHB
4      {0xe0000000, 0xe0040000}  // PPB
5  };
6
7  for (uint8_t i = 0; i < 3; i++) {
8    for (uint32_t addr = ranges[i][0]; addr < ranges[i][1]; addr += 512) {
9      *(uint32_t *)addr = *(uint32_t *)addr;           // WRITE
10     printf("0x%lx: %lx\n", addr, *(uint32_t *)addr); // READ
11   }
12 }
```

Listing 5.3: RIOT peripheral memory walk

Ultimately, we found user applications in RIOT may directly access any valid range of peripheral memory without restriction, including those requiring privileged execution; highlighting the fact that RIOT currently does not enforce mediation over peripheral access and that user application code runs with full system privileges.

RAM Access. We next turn our attention to Listing 5.4 below, finding user applications code may read, write, and execute any available RAM address, including that used by the kernel and other threads. The only found exception is the first 32-bytes of RAM, which represent the top of the stack (near the boundary to Flash memory). This exception is specific to RIOT supported ARM-based devices containing an MPU. RIOT uses the MPU to set this 32-byte region as read-only, resulting in our write and execute attempts on line 5 and 6 respectively to trigger a fault. This mechanism primarily exists to prevent accidental writes beyond RAM, for example, by an accidental infinite recursive call. Thus, with this small exception, all RAM can be read, written to, or executed in RIOT, at least by default (Section 5.3).

Furthermore, any MPU-based defense mechanism specific to RIOT is ineffective at defending against untrusted applications (e.g., those assumed malicious by default). This is due to the fact that user applications run with the same privileges as the kernel and may disable the MPU at any time (through direct peripheral access).

```
1  // Add 32 to skip MPU_STACK_GUARD protection
2  uint32_t ram_start = 0x20000000 + 32;
3  uint32_t ram_end = 0x20010000;
4
5  for (uint32_t addr = ram_start; addr < ram_end; addr += 512) {
6    *(uint32_t *)addr = 0x46f7;                          // WRITE
7    __asm__("blx %0" : : "r"(addr | 1));                 // EXECUTE
8    printf("0x%lx: %lx\n", addr, *(uint32_t *)addr); // READ
9  }
```

Listing 5.4: RIOT RAM memory walk

Notably, the ability for user code to alter any RAM value, including those used by the kernel, alludes to a lack of memory isolation and implies user application code must be modeled with the same level of complete trust required by the kernel.

Flash Access. Next, Listing 5.5 shows how a RIOT user application may directly read, write[2] and execute from any arbitrary Flash memory address. This includes memory where currently executing (kernel and application) code could exist, which in our case is the first two pages of Flash[3].

[2]Note, for direct alteration of Flash memory we mean that there are no restrictions on directly accessing the peripheral used to program Flash.

[3]Although writing to the first two pages is successful, our CPU does fault sometimes; likely due to erasing/writing Flash we are trying to immediately start execution of. However, we confirm our changes are indeed written through a debugging session.

```
1  static uint8_t PAGE_MEM[FLASHPAGE_SIZE] ALIGNMENT_ATTR;
2  uint32_t flash_start = 0x00000000 + FLASHPAGE_SIZE * 2;
3  uint32_t flash_end = 0x00080000;
4  uint8_t page;
5
6  for (uint32_t addr = flash_start; addr < flash_end; addr += FLASHPAGE_SIZE) {
7    page = flashpage_page((void *)addr); // Get Flash page where t() is stored
8    flashpage_read(page, PAGE_MEM); // READ Flash page containing t()'s code
9    *(uint32_t *)PAGE_MEM = 0x46f7; // Overwrite 4-bytes in the RAM page
10   flashpage_write_page(page, PAGE_MEM); // WRITE altered page back to Flash
11   __asm__("blx %0" : : "r"(addr | 1));  // EXECUTE
12   printf("0x%lx: %lx\n", addr, *(uint32_t *)addr); // READ
13 }
```

Listing 5.5: RIOT Flash page walk

The implications of the above show that an untrusted or attacker controlled user application may make persistent changes to both the application and the underlying OS that are maintained (and potentially executed) upon system power/reset.

Assessing E1 on Tock. In Tock's multiprogramming model, user applications are compiled and flashed (at distinct locations) independently of the kernel. Execution starts at the kernel which locates any applications in Flash then decides whether to load them as processes. Rather than direct function invocations, processes must trigger a service call (`svc`) interrupt (a.k.a., *system call*) to request kernel services [192]. Making a system call causes a processor context switch from the unprivileged mode (in which processes run) to privileged mode (in which the kernel runs). This allows the kernel to restrict access to resources, such as memory, for unprivileged processes but regain full control over all system resources when servicing a system call (or any other exception/interrupt). Thus, experiments on Tock may reveal distinct bound-

aries between memory explicitly allocated to the kernel and user applications.

Peripheral Access. Later in Listing 5.12, we will see a function `led_on()` (on line 7) that inevitably must eventuate in access to MMIO (to turn an LED on). Given in Listing 5.6, the underlying implementation of `led_on()` reveals a call to `command()`, a wrapper function for triggering *command*, one of Tock's seven system calls [196]. This layer of indirection foreshadows that a process can only access peripherals through kernel mediation.

```
1 int led_on(int led_num) {
2   syscall_return_t rval = command(DRIVER_NUM_LEDS, 1, led_num, 0);
3   return tock_command_return_novalue_to_returncode(rval);
4 }
```

Listing 5.6: Tock `led_on()` implementation

Through code identical to that shown in Listing 5.3, we test directly reading, writing, and executing a peripheral address from within a Tock user application. Each operation at the start of the memory walk immediately resulted in the OS kernel's hard fault handler being triggered due to a (MPU) memory access violation. Due to this prevention of direct access made from a user application to peripheral memory, kernel-mediated systems calls (such as that underpinning Listing 5.6) serve as the only viable method for user code to access peripheral memory. Discussed below, hardware-based confinement is the primary mechanism preventing this direct access.

RAM Access. We initially test arbitrary access to RAM, made by a Tock user application, using code identical to that used for RIOT (Listing 5.4); finding that this

code immediately triggered a system fault as well. As discussed (Section 2.3.3), Tock utilizes PMP[4] to confine its processes (user applications scheduled for execution). As a result, processes are only allowed to access memory (i.e., RAM and Flash) explicitly belonging to them (the amount of which is established at application compile-time). The Tock kernel will ensure this holds true by dynamically configuring relevant PMP regions before yielding execution to any given process [189]. When PMP is enabled, any unprivileged access attempt not coinciding with an established PMP region is denied by default[5]. Thus, in addition to being denied execution of its own RAM, processes are denied any access beyond their allocated RAM, i.e., that which may be utilized by the kernel or other user applications. The only exception to this rule is memory used for IPC [189], which can be requested from the kernel for multiple processes to share.

[4] We use the more general term PMP here as Tock supports other architectures with the same protection capabilities [164, p.10-11]. Despite our experiments targeting ARM, high-level discussion here roughly generalizes over any Tock-supported architecture.

[5] We discussed this in Section 4.1.2 for the ARM MPU, however denying access beyond established regions is likely a general rule of thumb. For example, we see this rule specified by the RISC-V ISA as well [165, p.52].

```
1  uint32_t ram_start = tock_app_memory_begins_at();
2  uint32_t ram_end = tock_app_memory_ends_at();
3  uint32_t memory_limit = sbrk(0);
4
5  for (uint32_t addr = ram_start; addr < memory_limit; addr += 512) {
6    *(uint32_t *)addr = 0x46f7;                    // WRITE
7    printf("0x%lx: %ld\n", addr, *(uint32_t *)addr); // READ
8  }
9  /* INVALID: every line below will trigger a fault */
10 __asm__("blx %0" : : "r"(ram_start | 1)); // Execute denied by default
11 printf("After used RAM: %ld\n", *(int32_t *)(memory_limit + 0x1000));
12 printf("Before allocated RAM: %ld\n", *(int32_t *)(ram_start - 4));
13 printf("After allocated RAM: %ld\n", *(int32_t *)ram_end);
```

Listing 5.7: Tock application RAM memory walk

Shown in Listing 5.7, each process has a fixed amount of RAM, between `ram_start` and `ram_end` in our case. Access is only granted to what is needed. Access to heap memory (in RAM) is granted upon request, otherwise access to most of the unused space is denied. Any access violation, such as those between lines 10-13 will result in a PMP fault trapping to the kernel-owned hard fault handler due to an access violation. Notably, line 10 shows that execution of RAM belonging to the application itself is also denied default. Thus, due to kernel-configured[6] PMP permissions, our attempt to execute assembly code stored on the stack within a process' RAM causes an exception as well.

Flash Access. Access to Flash memory is similarly governed through Tock's configuration of PMP. A fixed amount of Flash memory is allocated for each application

[6] https://github.com/tock/tock/blob/master/kernel/src/process_standard.rs

(8192 bytes by default). Before loading an application as a process, the kernel will set appropriate PMP regions over Flash with read and execute permissions only. The deny-by-default PMP rule ensures applications are confined to only execute from their given regions. Recalling writing to Flash typically requires an additional hardware peripheral and that peripheral access is mediated by the kernel, writing/erasing Flash can only be achieved if made available by and requested from the kernel (through a system call) [189].

```
1  uint32_t flash_start = tock_app_flash_begins_at();
2  uint32_t flash_end = tock_app_flash_ends_at();
3
4  for (uint32_t addr = flash_start; addr < flash_end; addr += 4) {
5    printf("0x%lx: %lu\n", addr, *(uint32_t *)addr);
6  }
7  /* INVALID: every line below will trigger a fault */
8  printf("Before allocated RAM: %lu\n", *(uint32_t *)(flash_start - 4));
9  printf("After allocated RAM: %lu\n", *(uint32_t *)flash_end);
```

Listing 5.8: Tock application Flash memory walk

In Listing 5.8 above, we can see that an application can read (and is executing) Flash memory allocated specifically to it. Trying to access Flash before or after what has been allocated will result in a fault, as seen on lines 8 and 9. This ensures each application is confined to its own Flash and denied access to Flash belonging to the kernel or other applications.

5.2. Kernel Component Access (E2)

Moving away from user applications, we next focus on kernel components, which we model as untrusted (Chapter 3) due to their potential to introduce malicious code into the system or be exploited through interfaces exposed to user applications. The goal of this experiment is to assess whether kernel components are restricted in any way from accessing resources belonging to user applications or the rest of the kernel.

Background. To consider the importance of modeling kernel components as untrusted, we may look at a well-established IoC OS; Linux follows a *monolithic design*, in which the core kernel, device drivers, and system libraries are tightly integrated, written in a memory unsafe language (i.e., C), and execute in the same privileged mode. Historically, this design may have lead to the success of Linux [40]. However, the large trusted base incurred by its monolithic design has evidently resulted in an ever-growing attack surface [148, 40], granting attackers who may inevitably find and exploit flaws complete privileged access and the opportunity to crash the entire system.

Research and development into Linux [148, 106] and entirely newly IoC OSs [129, 42, 150, 52, 173] has continuously sought to minimize the trust base. Evidently, the benefits of a small trust base and resource isolation may also have particular relevance in the development of multi-tenant IoT OSs due to a more pronounced convergence and complexity relating to the predominantly multi-tenant OSs found within the IoC. However, an important aspect differentiating and potentially limiting IoT OSs

is hardware support. For example, RIOT supports many devices lacking support for PMP, thus any established trust model (Section 6.2) would differ depending on selected hardware. Identifying the applicability and trade-offs of security foundations may be critical for limiting access and trust required by any system component.

Setup. This experiment closely resembles that of the previous experiment. The primary differentiation is the focus on kernel components, more specifically RIOT *drivers* and Tock *capsules*. By developing kernel components respective to each OS, we similarly will try to conduct a memory walk, testing for the possibility and any subsequent outcomes of each operation (read, write, execute) used to access addresses within the memory ranges identified earlier.

Assessing E2 on RIOT. The amount of access observed for user applications in RIOT alludes to the fact that kernel components may be granted with the same amount. In Listing 5.9, we create a kernel driver `foo` defining several functions, each of which executes code similar to listings given in E1, as noted in each respective comment.

```
#include "foo.h"
...
void foo_peripheral(void) {
  // Listing 5.7
}

void foo_ram(void) {
  // Listing 5.8
}

void foo_flash(void) {
  // Listing 5.9
}
```

Listing 5.9: Kernel driver access in RIOT

Any function from `foo` invoked by a user application using the driver leads to same outcomes observed for user applications in E1. Moreover, similar to user applications, (untrusted) kernel components may freely configure/disable MPU protections due to their privilege and unrestricted access.

Assessing E2 on Tock. Discussed in Section 2.3.3, the Tock kernel is written in Rust. We noted that unlike the rest of the kernel, Tock kernel components (a.k.a., capsules) are *sandboxed* by Rust's type-system. Software sandboxing is achieved in Tock by denying capsules the ability to use any code marked with the `unsafe` keyword. Previously highlighted in Listing 4.5, confining code to only use safe Rust eliminates every direct method we have used thus far to access memory. It further works towards eliminating a majority of software-based flaws, in turn greatly reducing the potential for capsules to cause a fault. However, there are certain aspects Rust's type-system does not wholly prevent, such as endlessly recursive functions or

benign stack-based resource allocation. Such scenarios do not break memory safety, but may exhaust available RAM or potentially cause the entire system to crash.

In Listing 5.10, we include a contrived example capsule for context. Each capsule, including our example, are developed in a Rust *library crate* (a collection of code modules to be consumed by others), where denying unsafe code is achieved via the crate-level attribute `#![forbid(unsafe_code)]`. Omitting this attribute would allow capsules to use unsafe code and break the compile-time safety guarantees protecting the rest of the system, making it paramount to capsule memory safety guarantees.

```
use kernel::hil::led;
use kernel::{..., Driver};
...
pub const DRIVER_NUM: usize = driver::NUM::Foo as usize;

pub struct FooDriver<'a, L: led::Led> {
  led: &'a L,
}

impl<'a, L: led::Led> FooDriver<'a, L> {
  pub fn new(led: &'a L) -> Self {
      led.init();
      Self { led }
  }
}

impl<L: led::Led> Driver for FooDriver<'_, L> {
  fn command(...) -> CommandReturn {
      match command_num {
        0 => { // Toggle
            self.led.toggle();
            CommandReturn::success()
        },
        1 => { // Recurse
            fn recurse() {
             let data = [1000; 0xff];
             recurse();
            }
            recurse();
            CommandReturn::success()
        },
        _ => CommandReturn::failure(ErrorCode::NOSUPPORT),
      }
  }
}
```

Listing 5.10: Tock capsule example

Once compiled into the kernel, user applications may invoke our example capsule

by triggering a `command` system call with the correct driver number and related arguments. A `command` call to our capsule with a command number of 1 will trigger an unconditional recursive call, seen between lines 24-29, causing a system-wide crash due to a kernel stack overflow (resulting from resource exhaustion); calling with command 0 will toggle a MMIO GPIO pin.

Important to note however, is that sandboxed capsules do not directly define the memory (e.g., MMIO) locations they will be accessing (as that would require unsafe code), instead, they accept a reference to a *hardware interface layer* (HIL) type. A HIL is defined by trusted kernel code and provides a standardized API over various hardware devices, such as chip-specific peripherals and off-chip hardware (e.g., sensors, radios, etc) [194]. The HIL provides traits and types that eventually get implemented and utilized by other trusted parts of the kernel, such as a specific chip. Thus, although a majority of the driver logic may be implemented within an untrusted capsule, the correctness of an inevitable memory access fundamentally relies on the trusted code providing the underlying HIL implementation.

Moreover, because capsules execute within the same single threaded cooperatively scheduled loop as the rest of the kernel, there is currently no way to preempt a long-running capsule. Thus, capsules may prevent the availability of other system components (such as user applications).

Overall, despite compile-time sandboxing of Tock capsules, memory safety guarantees provided by Rust itself may not necessarily provide the same guarantees estab-

lished by hardware-based confinement, at least with respect to security. However, language-based memory safety evidently prevents a large subset of programming errors, particularly those related to arbitrary memory access.

5.3. Initial Access for Landing an Attack (E3)

Building on knowledge about what can be accessed by applications and OS components, we next illustrate how an attacker may gain control of a benign, yet vulnerable, user application through the exploitation of a stack-based buffer overflow. The vulnerable applications contrived for this experiment aim to represent how exploitation can be used to gain initial access. This access can itself result in access violation, but will also set the stage for further subsequent experiments that better assess the utility and pronounce the importance of security foundations. This experiment will also allow us to test for the existence of any preliminary defense mechanisms working against stack-based buffer overflows.

Background. In general, every application must be granted a section of memory to store local data and save CPU *registers*. Consider execution of non-inlined[7] functions, where a *caller* branches to the memory address of a function (a.k.a., the *callee*). The local *scope* of the callee will be established through a series of initial machine instructions (as part of the callee code) to save any registers the callee will manipulate during execution and/or restore once done execution.

[7] During compilation, some simpler functions may get *inlined*. This is an optimization that directly substitutes the function definition, avoiding extra overhead required to set up a new unnecessary stack frame.

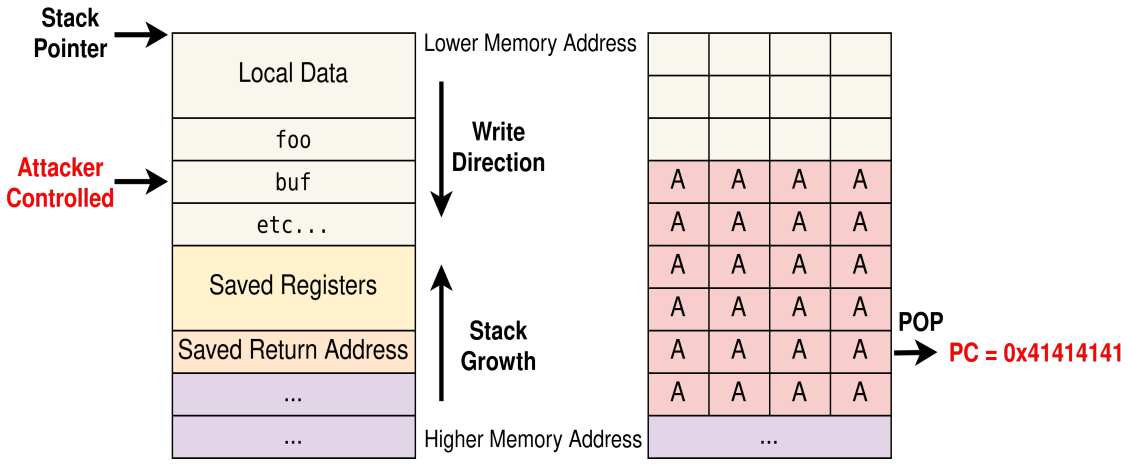

Figure 5.2: Generic stack overflow example

These registers are saved into an extensible but finite memory region in RAM called the *stack*. As depicted on the left of Figure 5.2, the beginning of stack memory is typically placed at a higher memory address and grown towards lower memory addresses. For a target register to be saved onto the stack, a register called the *stack pointer* (SP) is decremented to grow the stack, then the value held in the target register is stored in the newly created space. For example, when a function is called, the first few instructions within the function, called the function *prologue*, typically store a register referred to as the *frame pointer* (FP) onto the stack. The scope of a function is defined through the combination FP and SP, as they can be used in tandem to define stack memory used by the current function (a.k.a., the *stack frame*). In addition to registers, the stack may be used to save local data, simply by growing the stack and maintaining a pointer to the memory containing the saved data.

When a function is called, the return address of the caller (i.e., the instruction right

after the branch instruction) is automatically placed in a special register called the *link register* (LR). If the callee intends to return back to the caller, it stores LR onto the stack during its prologue then, when done, returns to the caller by loading the saved LR into the *program counter* (PC) register during its *epilogue*. Importantly, the PC register always holds the next instruction for the CPU to fetch then execute. The epilogue will restore the stack by tearing down the stack frame, then return by popping LR off of the stack into PC. The fact that LR is first stored onto the stack during the function prologue, means it always remains below (at a higher memory address) local data subsequently stored. Consequently, a vulnerability may manifest when a stack-based buffer is mismanaged by software. If data is allowed to be written past a buffer's allocated space, an overflow will occur that could potentially alter the saved LR, resulting in an altered/overflown value being loaded into PC.

Setup. In this experiment, we assume the role of an attacker with the ability to interact with our test device. We create two user applications, one for RIOT and one for Tock, each of which contain a stack-based buffer overflow vulnerability and also accept arbitrary user input over USB. To exploit each vulnerable application, we construct adequately sized strings and send them over USB to cause an overflow. We show how such overflows may be used to redirect normal execution flow, an aspect further expanded upon in subsequent experiments. Finally, we use this chance of altered execution to toggle a physical LED on our device, highlighting the fact that vulnerabilities on low-end IoT devices may lead to physical consequences (imaginably more severe than a toggled LED).

Assessing E3 on RIOT. The C code in Listing 5.11 is a RIOT application made vulnerable to a stack-based overflow. The `fread()` function accepts arbitrary user input over USB and stores the input in `DATA`, a buffer of size 60. The `vulnerable()` function copies all 60 characters within `DATA` into `buf`, which is a buffer of size 50 stored within `vulnerable()`'s established stack frame. The size difference results in an overflow extending beyond space originally allocated by `vulnerable()`'s prologue. In the absence of the stack overflow, execution would normally return back within the `while` loop. However, the overflow in this example writes far enough to alter the saved return address (recalling Figure 5.2). In the absence of any security mechanism(s), the result of this alteration will lead to unintended execution. To illustrate how an attacker may exploit this vulnerability, we have included a `target()` function that is never invoked, but will turn on an on-board LED if invoked.

```c
#include "periph/gpio.h"

static char DATA[60];

static void target(void) {
  gpio_clear(18); // Turn on LED2 (active low)
}

static void vulnerable(void) {
  char buf[50];
  memcpy(buf, DATA, sizeof(DATA));  // Overflow occurs here
}

int main(void) {
  // Does nothing, but ensures target() does not get optimized out
  void __attribute__((unused)) (*dummy)(void) = &target;

  gpio_init(18, GPIO_OUT);
  while (1) {
    fread(DATA, sizeof(DATA[0]), sizeof(DATA), stdin);
    vulnerable();
  }
  return 0;
}
```

Listing 5.11: RIOT user application vulnerable to stack overflow

Sending 60 bytes will cause `fread()` to complete and trigger the buffer overflow. Before we trigger the overflow, we first utilize a debugger to quickly determine the (4-byte) address of `target()`. Then, we send a sequence consisting of this address[8] repeated 15 times. Consequently, `vulnerable()` will be called, causing an over-

[8]For the address to be correctly loaded by our device's CPU, we must take into account two specific conditions. First, data accesses on our ARMv7-M-based device are always interpreted in little endian format [19, p.68]. Second, the ARMv7-M instruction set also requires addresses loaded into PC have a LSB equal to 1 [19, p.125]. For example, if we wanted place the address $0x000000dc$ onto the stack, we must send the escaped hex string $\backslash xdd\backslash x00\backslash x00\backslash x00$, noting the correct byte order and LSB.

flow by copying our sequence of return addresses from `DATA` into `buf` and beyond. Once complete, `vulnerable()` will return through its epilogue, loading the address of `target()` into PC. This successfully completes without any mechanism in place to stop it, turning on the on-board LED as a consequence.

Interestingly, RIOT contains an optional system library[9] introducing the *stack smashing protector* (SSP) [141] compiler feature. The basis of SSP is predicated on automatic code transformations, where additional code is inserted into relevant non-inlined functions at compile-time. As depcited in Figure 5.3, a value called a *stack canary* will first be placed on the stack right above any registers saved during the prologue. Then, additional code is interposed between the function body and epilogue that checks the integrity of the canary. If an overflow were to occur, the canary value would always be overwritten first and before saved registers, including the return address. Subsequently, the integrity check would fail and allow for a decision to be made by the kernel developers. RIOT's SSP library triggers a panic causing a hard fault function handler to be invoked. Once trapped to the hard fault handler, architecture specific code handles the access violation. RIOT's hard fault handler by default prints relevant debug information then resets the entire system.

[9] https://doc.riot-os.org/group__sys__ssp.html

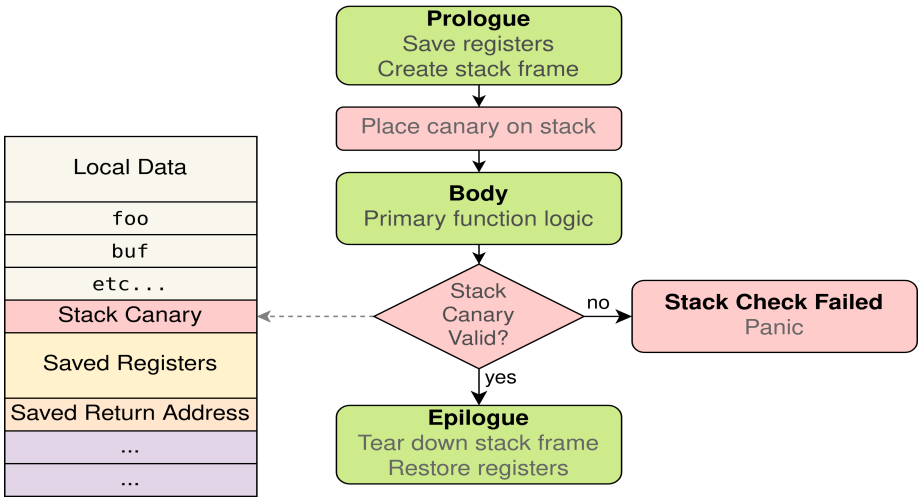

Figure 5.3: Stack canary flow diagram

In RIOT, the canary value is a random 64-bit number that remains the same for a specific firmware image. The value is updated when code is recompiled and flashed onto the device. Further, despite the canary being 64-bit, we found through our testing that only a single machine *word* is stored and validated. In our case, we are testing on a 32-bit word size device, thus only the least significant 32-bits of the canary are stored on the stack for subsequent validation. An attacker must determine the canary (a 32-bit entropic value in our case) to bypass this mechanism. To test this, we enabled SSP, stepped through a debugging session to learn the current firmware's canary value then, instead of overflowing the stack completely with our desired return address, we sent repetitions of the canary value followed by our desired return address[10]. The result successfully bypasses SSP, allowing us to alter the return value as before. Overall, the effectiveness of this mechanism relies on keeping the

[10]We also had to increase the size of `DATA` (Listing 5.11) from 60 to 64. This accounted for the stack increase needed to store the canary and allowed our overflow to reach the saved return address.

canary secret, which, under our threat model, leaves bruteforce guessing [35] as the most evident threat. Notably, this finding is not wholly relevant to our discussion on foundations but nonetheless shows that there are additional defense mechanisms viable for low-end devices not requiring substantial/any code refactoring that can be added later on in the developmental life cycle.

Assessing E3 on Tock. The same principles described in the RIOT example above similarly apply to Tock applications written in C, such as that of Listing 5.12 below. The `getnstr()` function accepts arbitrary user input over serial USB, resulting in a stack overflow.

```c
#include <led.h>

static char DATA[60];

static void target(void) {
  led_on(1); // Toggle on LED2
}

static void vulnerable(void) {
  char buf[50];
  memcpy(buf, DATA, sizeof(DATA)); // Overflow occurs here
}

int main(void) {
  // Does nothing, but ensures target() does not get optimized out
  void __attribute__((unused)) (*dummy)(void) = &target;

  while (1) {
    getnstr(DATA, sizeof(DATA));
    vulnerable();
  }
  return 0;
}
```

Listing 5.12: Tock user application vulnerable to stack overflow

Through a GDB session, we determine the address of the uncalled function is $0x00030096$. Sending the appropriately formatted repeated sequence of this address successfully invokes `target()`, resulting in the on-board LED turning on. Unlike RIOT, we found no optional (or mandatory) stack canary implementation for Tock.

5.4. Payload Injection (E4)

Building directly on E3, we next examine whether an attacker may take advantage of stack-based overflows to inject and execute arbitrary code on the stack. The goal is

to determine whether this form of arbitrary code execution is possible or to identify the existence of any preventative mechanisms.

Background. Written code must eventually be turned into *machine code*, sequences of binary data understood and executed by a specifically targeted CPU. This binary format (i.e., machine code) is specified by the CPU's underlying ISA, which also specifies a corresponding *assembly language* serving as the lowest human-readable interface with a one-to-one mapping to machine code. These days, assembly code is the final intermediary representation produced by a higher-level programming language (e.g., C, Rust, etc) compiler. Compilers typically support many different ISAs, allowing them to produce assembly code to be subsequently turned into machine code by an *assembler*.

```
1  @ This is a comment
2  @ assembly code         machine code     pseudocode
3  @ --------------------  ------------     ----------
4    mov r4, #5            @ 4f f0 05 04    r4 = 5
5    mov r1, r4, lsl #28   @ 4f ea 04 71    r1 = 5 << 28 = GPIO base address
6    mov r2, r4, lsl #8    @ 4f ea 04 22    r2 = 5 << 8
7    add r2, r2, #4        @ 02 f1 04 02    r2 = r2 + 4 = GPIO OUT port offset
8    add r1, r1, r2        @ 11 44          r1 += r2 = GPIO OUT address
9    ldr r2, [r1], #4      @ 51 f8 04 2b    r2 = value stored in GPIO OUT
10   mov r3, #1            @ 4f f0 01 03    r3 = 1
11   lsl r3, r3, #17       @ 4f ea 43 43    r3 = r3 << 17 = LED1 bit number
12   eor r2, r3            @ 82 ea 03 02    Toggle LED1 bit on
13   str r2, [r1, #-4]     @ 41 f8 04 2c    Write result back to GPIO OUT
```

Listing 5.13: ARM assembly for toggling a GPIO pin

In Listing 5.13, assembly code is displayed beside its corresponding machine code (in hexadecimal format) targeting an ARM Cortex-M4 [22] CPU implementing the

ARMv7-M ISA [19]. The code reads in the memory-mapped address of a general-purpose input/output (GPIO) port pertaining to an nRF52832 MCU [135], then writes back the value with a specific bit-flipped to toggle a physical pin connected to an LED on our chosen test device. As discussed, instructions are normally stored in Flash memory, yet theoretically, execution of these instructions may be achieved by storing them at a location on the stack then loading PC with the address of that specific location.

Setup. This experiment utilizes the same code from Listing 5.11 for RIOT and Listing 5.12 for Tock. Deciding to inject and execute the payload above in Listing 5.13, we collect the given assembly code as a string of escaped hex characters, as shown in Listing 5.14. To ensure this string is 60 characters long (enough to trigger the overflow), we pad it with several *no operation* (NOP) instructions (highlighted in green). NOPs are valid instructions which do not directly change any resource (e.g., memory or program-accessible register values), executing such instructions will simply result in PC being incremented to allow the next instruction to execute. We follow the NOPs with the actual assembly instructions (in blue) we wish to execute. The final four bytes (in red) comprise the return address used to redirect execution flow (as done in E3), however, instead of jumping to a function like before, this address will instead be chosen to point back into the stack (56 bytes behind the saved return address) to where our NOPs and assembly instructions exist.

```
\x09\x46\x09\x46\x09\x46\x09\x46\x09\x46\x09\x46\x09\x46\x09\x46
\x09\x46\x4f\xf0\x05\x04\x4f\xea\x04\x71\x4f\xea\x04\x22\x02\xf1
\x04\x02\x11\x44\x51\xf8\x04\x2b\x4f\xf0\x01\x03\x4f\xea\x43\x43
\x82\xea\x03\x02\x41\xf8\x04\x2c\xFF\xFF\xFF\xFF
```

Listing 5.14: Attack string containing 9 NOPs (green), 10 assembly instructions of interest (blue), followed by the return address (red) to be replaced

Previously in E3, we found initial access through injection was possible under certain conditions for both RIOT and Tock. Thus, this experiment focuses on whether altering the return address to point to a payload on the stack (located in RAM) may result in code execution.

Assessing E4 on RIOT. Starting with RIOT, we send the attack string outlined in Listing 5.14 to the vulnerable program (with SSP disabled), appropriately replacing the return address (`\x2d\x08\x00\x20` in our specific setup) to point back to our code on the stack. The result is successful code execution leading to the on-board LED being turned on. However, we must be quick to note this is as the default behavior in RIOT.

Alternatively, RIOT provides an optional feature[11] that can be enabled at compile-time to make RAM non-executable (i.e., read-write only). This feature is only available on certain ARM Cortex-M devices (such as our test device) due to reliance on an MPU (Section 4.1.2). With this feature enabled, an MPU region is set up by the kernel during system initialization to ensure non-executable RAM. Notably, making

[11] https://github.com/RIOT-OS/RIOT/blob/master/cpu/cortexm_common/vectors_cortexm.c#L149

RAM non-executable prevents any code execution in all RAM-allocated sections, including the stack and heap. This also applies to kernel code, as both user applications, kernel components, and the core kernel differ very little from one another upon execution. This means users deciding to opt-in for non-executable RAM (assuming support in hardware) will also prevent kernel code from executing code in RAM (as long as all components refrain from disabling the MPU at runtime).

Repeating our experiment with this feature enabled results in the overflow writing over stack data as before, but any subsequent execution of the assembly code being denied. Instead, an MPU access violation occurs resulting in the OSs hard fault function handler resetting the entire system. Overall, providing a non-executable stack on platforms supporting PMP (such as those with an ARM MPU) may make exploitation relying on execution in RAM non-trivial to exploit, instead requiring more advanced techniques not reliant on code injection in RAM (Section 5.6).

Assessing E4 on Tock. We repeat this experiment on Tock, sending the same string given in Listing 5.14, only replacing the return address (in red) with an appropriate address within the stack. As we know, Tock utilizes hardware-backed PMP by default, setting RAM as read-write only. Thus, our targeted attempt to execute assembly code stored on the stack within a process' RAM causes an exception that traps to the trusted kernel's hard fault handler.

Interestingly, Tock additionally provides an optional feature to gracefully recover

from faults triggered by an application. Through a set of *fault policies*[12], the Tock kernel can be configured to, for example, stop or restart a faulting process, where the former provides fault isolation and the latter goes one step further to also allow fault recovery for processes (c.f. Section 6.5).

5.5. Maneuvering Around Defense Mechanisms (E5)

When assessing E1 on RIOT, we found it possible for any code on the system to directly access peripheral memory, even those specified as private (i.e., only accessible in the privileged mode), due to lack of privilege separation. After that in E4, we saw that arbitrary code execution on the stack may be thwarted by hardware-based PMP protections in Tock and in RIOT when enabled. By considering the outcome of these two prior experiments and the fact that PMP is configured through private registers (c.f. Section 4.1.2), this experiment aims to examine the role and importance of privileges modes as a foundation for systems relying on PMP for access control.

Background. As we have seen, setting the stack as non-executable effectively mitigates against arbitrary stack-based code execution. However, it does not prevent an attacker from diverting control flow, as described in E3 and experimented with again in E4. Thus, if given the opportunity, an attacker may still divert control flow to any valid code location(s) (e.g., in Flash memory) directly accessible by the target. In general, this process is typically referred to as a *code reuse* [186, 166] attack. In this

[12] https://github.com/tock/tock/blob/master/kernel/src/process_policies.rs

experiment, we directly build upon prior experiments and ideas relating[13] to code reuse to examine the importance of privilege modes for systems relying on PMP.

Setup. We seek to inject and execute code on the stack, even after the target system has already configured the stack as non-executable via PMP. To approach this, we introduce slight modifications to code given in Listing 5.11 for RIOT and Listing 5.12 for Tock, outlined respectively below. We include functions that, if invoked, may allow code to disable PMP, but only if said code is running in the privileged mode. We do not plan to invoke these functions directly in our modified programs, instead we will attempt to modify and inject the attack string from Listing 5.14 to conduct a code reuse attack. This will involve redirecting control flow to the aforementioned function capable of disabling PMP. Subsequently, our attack string must also cause a secondary redirection to the instruction payload which, upon execution, will visually indicate success via an LED as before. If our attack succeeds, this experiment will have demonstrated how an attacker may conduct code reuse attacks to fully disable PMP on systems relying on PMP but lacking adequate privilege separation. As a result, attackers may be capable of further exploitation by gaining back the ability to inject and execute arbitrary instructions of their choosing on the stack. If the attack fails, we hypothesize the root cause may be due to the utilization of privilege modes, used to prevent unprivileged code from accessing PMP registers. We hope to conclude that utilizing the foundation of privilege modes is critical to ensuring configured PMP protections remain unaltered by code ideally modeled as untrusted (Section 3.1).

[13] We discuss more specific and advanced code reuse techniques in Section 5.6.

Assessing E5 on RIOT. Starting with RIOT, Listing 5.15 introduces a similar but slightly modified version of that given in E3. The `vulnerable()` function remains the same, however we enable the optional feature found in E4 to make RAM (and thus the stack) non-executable. Enabling this feature does not require users change their application source code[14], but if users want to enable, modify, or disable PMP at runtime themselves, they may include the appropriate OS provided code to do so; our primary code modifications to Listing 5.15 involve including this code. As our test device is ARM-based, we include the header for configuring the MPU on line 1. Due to compiler optimizations enabled by default in RIOT, any unused functions (including those from the header) get compiled out. Thus, on line 11 we create a function pointer pointing to a function of particular interest from the header, that is, one for disabling the MPU entirely[15]. This pointer ensures said function is not compiled out, allowing the function to exist in the code but remain uncalled. Just as in E4, if we were to now inject the prior attack string onto the stack, execution would be denied, resulting in a system-wide reset.

[14]The feature is gated by a compiler flag which users may opt-in through their project's Makefile.
[15]https://github.com/RIOT-OS/RIOT/blob/master/cpu/cortexm_common/mpu.c#L24

```c
#include <mpu.h>

static char DATA[60];

static void vulnerable(void) {
  char buf[50];
  memcpy(buf, DATA, sizeof(DATA));
}

int main(void) {
  // Does nothing, but ensures mpu_disable() does not get optimized out
  int __attribute__((unused)) (*dummy)(void) = &mpu_disable;

  while (1) {
    fread(DATA, sizeof(DATA[0]), sizeof(DATA), stdin);
    vulnerable();
  }
  return 0;
}
```

Listing 5.15: Modified version of Listing 5.11

However, if, through additional means, an attacker were able to disable MPU protections, they may freely inject and execute code on the stack thereafter. In our contrived example, code for disabling the MPU has been provided as mentioned. Our goal then, is to divert control flow to reuse said code, then, to subsequently coerce execution of our chosen assembly instructions placed on the stack right after. By making modifications to the attack string from E4 (Listing 5.14) we achieve exactly this. We defer the specific details to Appendix A. In summary, our modifications allow us to first redirect control flow to invoke `mpu_disable()` then subsequently redirect control flow a second time back to the assembly instructions in our attack string. The result shows that, without privilege separation to protect private registers in RIOT, any code running as part of a RIOT system can disable PMP protections.

Disabling PMP leaves a RIOT system more vulnerable, as attackers can execute straightforward code injection attacks with greater ease; potentially suggesting that OSs (and any involved entities) should not expect PMP security guarantees to be upheld if not used in tandem with privilege modes.

Assessing E5 on Tock. As we learned in E1, Tock user applications are denied access to any peripheral memory which, if we recall, includes PMP registers. Thus precluding any access by user applications to PMP registers. To test this in the context of this experiment, we provide Listing 5.16 as a modified version of that previously given in E3. As user applications presumably cannot access PMP registers, there is no provided user library providing a function equivalent to RIOT's `mpu_disable()` we can target. Thus, in lines 3-6 we provide our own equivalent[16].

[16]Recalling Section 4.1.2, the `MPU_CTRL` register is memory-mapped to address `0xE000ED94` and refers to the MPU Control Register of an ARM-based device's MPU [22, p.59]. The first bit of this register is the `ENABLE` bit [19, p.693]. If set to 1, the MPU is enabled, if 0, it is disabled. Code in Listing 5.16 clears the `ENABLE` bit using a bitwise operation, if successfully executed this would disable the MPU.

```
1  static char DATA[60];
2
3  static void mpu_disable(void) {
4    uint32_t *MPU_CTRL = (uint32_t *)0xE000ED94;
5    *MPU_CTRL &= ~0x1;
6  }
7
8  static void vulnerable(void) {
9    char buf[50];
10   memcpy(buf, DATA, sizeof(DATA));
11 }
12
13 int main(void) {
14   mpu_disable();
15   while (1) {
16     getnstr(DATA, sizeof(DATA));
17     vulnerable();
18   }
19   return 0;
20 }
```

Listing 5.16: Modified version of Listing 5.12

We forgo injecting an attack string to alter control flow for invoking `mpu_disable()`, instead opting to simply invoke it directly on line 14. We chose this simplification as our initial debugging sessions confirm that execution proceeds only up until `mpu_disable()` is invoked. Once invoked, a fault is triggered by the access and the application is terminated. With a bit more explaining, we can eliminate the need for continuing further with a full-fledged attack. To start, this outcome is a result of several factors. First, as we learned, Tock utilizes PMP for more than setting RAM as non-executable; it additionally confines user applications to pre-established regions of memory granted to that application. This means the observed fault was not generated due difference in privilege, but due to the dynamically allocated PMP

established by privileged kernel code prior to the application being scheduled. However, even if PMP was not used to confine Tock user applications, applications would still be denied from accessing PMP registers. We can test this by disabling PMP[17] then examining execution of the code in Listing 5.16 again. Doing so results in another fault occurring, this time due to the unprivileged user applications attempting to access a register only accessible by privileged code. Overall, this highlights how correct usage of privilege modes to separate out code may prevent tampering of security critical resources.

5.6. Tabulated Summary and Discussion

Experiments conducted throughout this chapter examined the utilization of foundational security mechanisms employed by two actively developed IoT OSs. During these experiments, we also identified additional non-foundational mechanisms, such as the stack canary. Experiments were made to focus on access to memory-mapped private registers and peripherals, RAM, or Flash memory, which when taken together form the single physical address space of a low-end device. As every software component comprising an IoT OS shares this single physical address space, our focus on memory access gave insight into the amount of access components we modeled as untrusted have and exactly how security mechanisms may be used to prevent access violations originating from said components.

[17] Disabling PMP in Tock can be achieved by removing the line referenced by the following: https://github.com/tock/tock/blob/1.4.1/kernel/src/sched.rs#L268.

As summarized in Table 5.1, the foundations/mechanisms utilized by each system varies quite significantly. Further, even though mechanisms such as PMP were utilized between both systems, the specific details of such utilization differ, notably, in ways affecting security. To better understand why, we may recall the difference between *foundation* and *mechanism*. That is, a security foundation was said to be a mechanism, but one which typically requires tight integration during the initial developmental stages of an OS to effectively uphold security guarantees (c.f. Chapter 4). In general, foundations and standard mechanisms may be critical to upholding system properties relating to security, but foundations are those which a system adapted from the beginning and took into account the necessary design considerations needed to utilize the underlying mechanism to its full potential (and thus have it established as a foundation). As mentioned, this distinction highlights how foundations adhere to the principle of security by design. We next recap our findings on RIOT and Tock, then revisit the importance of foundations thereafter.

Resource		RIOT		Tock	
		User Applications	Kernel Components	User Applications	Kernel Components
Private Registers	R/W				
Peripherals	R/W				
RAM	R				
	W				
	X				
Flash	R/W/X				

Table 5.1.: Security mechanisms employed against unwarranted memory access violations in selected platforms on our test device; blank entries denote no employed mechanism (full unrestricted access)

Hardware-based:  physical memory protection,  privilege modes

Software-based:  language-based sandboxing,  stack canary

Focusing on RIOT first, we found in E1 and E2 that RIOT does not employ any mechanism to prevent access to private registers, peripherals, RAM, or Flash. However, subsequently in E3 we found RIOT does contain an optional stack canary mechanism to help mitigate against attackers maliciously diverting control flow via stack-based overflow vulnerabilities. Later in E4, we found RIOT has another optional feature for establishing PMP rules to prevent execution on the stack.

Next, experimenting with Tock gave us a hands-on experience with the mechanisms we had first learned it supports during our overview (Section 2.3.3). In E1,

we confirmed that user applications are restricted by PMP, and later in E5 we found upholding such restrictions were backed by privilege modes. More specifically, PMP restrictions denied user application memory access to any private registers, peripherals, and RAM and Flash not explicitly allocated to the application. Additionally, PMP rules are applied to the allocated memory of an application as well, leaving allocated RAM read-write only and allocated Flash read-execute only. As a result, applications are completely isolated from the rest of the system, only able to restrictively access allocated resources and having to request access to other resources via the OS kernel. Regarding the kernel, we examined in E2 how Tock kernel components (i.e., capsules) are isolated via language-based sandboxing mechanisms. Relying on Rust's memory safety, capsules are prevented from accessing any resource they are not meant to access at compile-time, providing a form of isolation; given the absence of flaws in core kernel-provided encapsulated unsafe code which certain capsules may rely upon.

The contrast between what and how mechanisms are applied in these two OSs starts to highlight the importance of security by design and its relation to security foundations. In particular, how PMP is tightly integrated (in tandem with privilege modes) in Tock exemplifies the definition of a foundation when contrasted with RIOT's use of it loosely as a mechanism. Tock is designed around PMP, its applications must be allocated and loaded into specific regions of memory, and the kernel must be aware of these regions to apply PMP protections. Further, Tock's applications run as unprivileged, providing defense-in-depth against access attempts to private registers and an environment in which the kernel may catch, isolate, and

recover from faults induced by any user application. RIOT on the other hand does not provide the same security benefits, PMP is not as integrated and privilege modes are not used. The consequence of this was examined in E5, where lack of isolation and access control allowed us to disable PMP entirely from a user application. This may allow us to conclude that because RIOT does not use PMP as a foundation, but rather as a mechanism loosely coupled with its execution environment, it becomes extremely vulnerable to attack.

However, using PMP as a foundation would require significant effort from RIOT developers; requiring a separation of kernel code and application code at the binary level at compile-time and potentially at the privilege level upon execution. This, in turn, would require implementing a system call interface between kernel/user code, as opposed the current direct function invocation interface. Moreover, such an implementation would likely incur overhead (for dynamic PMP configuration and context switching), potentially affecting RIOT's real-time capabilities, and only be applicable to devices with underlying support for PMP and multiple privilege modes.

Discussion thus far is not meant to paint a negative picture over RIOT, rather it serves to point out constraints introduced by low-end devices and the challenges they pose to OS security and portability. It may also allude to an increased relevance in software-based security mechanisms. For example, Tock's use of Rust as a foundation is a novel method to provide kernel component isolation with little overhead, and RIOT's optional stack canary is a simple, yet pragmatic, software-based mechanism that can be included without the need for an extensive code rewrite. As software-

based security mechanisms do not rely on underlying hardware support, they may be particularly relevant to IoT OSs aiming to support low-end devices lacking such underlying support.

Systems security is a complex problem, navigating around constraints imposed by low-end devices while also trying to satisfy diverse IoT use cases add to the challenge of this problem. However, considering the benefits and trade-offs of security foundations may be critical in adequately reducing the potential for access violations. Due to the nature of foundations, security considerations may be needed early on in the design or development stages of an IoT OS. In the next chapter, we shift discussion towards bringing several aspects relating to IoT OS security to the surface, looking to formalizing our work thus far and build upon it to identify criteria developers may work towards satisfying to improve the security of an IoT OS.

Chapter 6.

Criteria for Securing OSs Supporting Low-End Devices in the IoT

This chapter outlines a set of criteria towards securely developing (new or preexisting) IoT OSs supporting low-end devices. Additionally, we contextualize and compare these criteria against our two selected systems, RIOT and Tock. We split our criteria into two sets, foundational and supporting. Foundational criteria (F1–F5) serve to formalize and parallel our prior discussion and correlate with hands-on work in an independent fashion. Supporting criteria (S1–S4) also build upon our discussions, but primarily outline how developers may build above and maintain systems with established foundations that already adequately satisfy foundational criteria. Notably, if, upon examination, a system is found to inadequately satisfy certain

foundational criteria, it may suggest an inherent lack of security foundations, which fundamentally serve to prevent access violations. Thus, although both sets of criteria work towards enabling developers to more effectively develop, deploy, and maintain secure IoT OSs, foundational criteria ultimately hold precedence for establishing an initial secure execution environment suitable for subsequent development towards, potentially just as important, supporting criteria.

6.1. Small Trust Base (F1)

Maintain and guarantee the correctness of a small trust base that is resistant to tampering and which encapsulates all security-critical functionality

A small trust base may serve to establish hardware-based protections and encapsulate a majority of security-critical functionality requiring considerable access to underlying resources, e.g., direct memory access, scheduling, memory allocation, etc, while remaining minimal in terms of code size [140, p.22]. Importantly, an established small trust base should always be the first component to begin execution upon device power/reset and remain resistant to tampering thereafter. If lacking this initial base, an IoT OS may have no way to establish and subsequently enforce resource isolation (F3) and erroneously grant other components undue access to other parts of the system. Further, a small trust may allow an OS to establish its trust models (F2) through a small amount of, ideally easy to audit (S4), code that also encapsulates what is required to provide essential functionality (e.g., scheduling) and access underlying hardware (i.e., that which is inherently unsafe and typically *must* be trusted

by the rest of the system). As a result, a small trust base should reduce the attack surface on critical system components [40], grant an OS the ability to contain and recover from faults (F5), and safely extend trust outwards where appropriate [1].

Comparing RIOT to Tock on F1 RIOT is developed in a highly modular fashion, however this does not necessarily mean that upon execution RIOT has a small trust base. Referring back to Section 2.3.2, RIOT's core kernel includes support for scheduling, context switching, IPC, and synchronization primitives (e.g., mutexes, etc) [31]. All other components are developed separately from the core kernel, including drivers and hardware abstractions. However, upon execution, all this code is compiled together into a single monolithic binary (along with user application code) and executed with the same privileges (even on devices supporting multiple privilege modes, the consequences of which we discussed through our experiments). Further, the fact that RIOT is written in C, a type- and memory-unsafe language, means that RIOT and the applications developed atop it lack compile-time guarantees towards the reduction of software bugs/vulnerabilities.

Moving on, Tock follows a microkernel approach, only including essential functionality to support scheduling, context switching, IPC, system calls, interrupts, memory management, and hardware abstraction [105]. As discussed, Tock separates kernel code into trusted and untrusted code. Primarily achieved through the Rust programming language, Tock keeps its trust base small and uses it to encapsulate all required unsafe code, then integrate untrusted kernel capsule components that are sandboxed at compile-time by Rust's powerful type system (Section 4.2.2), all with

little resulting overhead [104]. Evidently, this allows Tock to avoid including large sections of what would be trusted code (e.g., device drivers) in their trusted kernel core, enabling efficiency and low-overhead typically seen in monolithic kernel implementations [105]. Finally, we also learned and examined through experimentation Tock's extensive use of multiple privilege modes and PMP to confine user applications, in turn preventing any unwarranted access to any part of the kernel (trusted or otherwise). By limiting the hardware it supports, Tock ensures these guarantees remain invariant.

6.2. Established Trust Models (F2)

Establish and document one or more model(s) to clearly capture system security guarantees currently provided, considering access capabilities/boundaries of any potentially involved entities and system components

Throughout their life cycle, IoT devices are subject to many different interactions and may be deployed in unexpected or hostile environments (Chapter 3). The various entities that develop, deploy, and use these systems should not be granted the same levels of unwarranted trust. Similarly, this also applies directly to software, where an OS's small trust base (F1) ideally remains as the only trusted component for upholding, establishing, and enforcing access control/isolation.

If adopting this criterion, the core developers of an IoT OS should establish a set of explicitly stated, accessible, and well-defined trust models, from the beginning [140,

p.24]. Upholding said models may require utilization of security foundations (Chapter 4) and satisfying additional criteria, such as resource isolation (F3) and fault isolation/recovery (F5). Establishing trust models may be of particular importance for multi-tenant OSs due to the fact that they may be utilized for hosting multiple components on a single device, each of which may be provided by different mutually distrusting parties [104]. Regardless, this criterion may serve any system (including those that are monolithic) for it may motivate reluctant allocation [140, p.24] of access privileges across components and outline, for any involved party, what explicit and (possibly unauthorized) implicit access boundaries exist between system components.

Comparing RIOT to Tock on F2 There is no clearly defined trust model for RIOT within the current literature [31, 32, 33] or its publicly stated design goals [157]. This lack of established trust models makes it difficult to ascertain what security guarantees RIOT provides to any involved parties, potentially leading RIOT developers and users to unknowingly downplay relevant security goals, particularly on systems running or making use of multiple potentially vulnerable user applications, system libraries, and kernel drivers.

On the other hand, Tock has been built from the beginning to support a multi-programming environment comprised of mutually distrustful components [104]. As discussed, Tock has a trusted core kernel and every component, process, and involved party must trust this base. Using component isolation techniques, Tock establishes a trust base that defines the trust model governing its user processes and untrusted

kernel components (capsules). Although each of these components have different goals, trust is reluctantly allocated to both in a relatively fine-grained manner allowing mutual distrust at the kernel and application level, guaranteeing interfaces to resources explicitly exposed by a resource owner are the only permitted interfaces to any resource in general [105]. Moreover, the different levels of trust between components is clearly outlined in an easily accessible public document [207] adjacent to the OSs actual source code.

6.3. Resource Allocation and Isolation (F3)

Allow for explicit allocation of device resources to software components, ensure resources granted to one given component remain isolated from any other unrelated components to prevent unwarranted access attempts, and enforce resource allocation limitations to prevent potential runtime resource overuse/exhaustion

Given code, data, and peripherals are all accessed through operations on memory, enforcing memory allocation and isolation restrictions is central to this criterion. Once explicit allocations can be made, isolation may be used to prevent access violations originating from kernel drivers to the core kernel, user applications to kernel components, and each application running in user space to any other part of the system (including other user applications). Utilizing a combination of hardware- or software-based foundations and ancillary mechanisms, an OS should work towards reducing the potential for any access violation and contain/recover from faults (F5), both of which may help achieve relevant security goals (Section 3.3) by removing the

possibility for compromised components to negatively affect other components or system runtime. Ideally, this should be achieved without unreasonable performance costs or restricting explicit communication capabilities between components (F4).

Among the constraints imposed on low-end IoT devices, energy may be one of particular importance to ensuring overall availability (c.f., Section 2.2.2). In addition to making use of deep sleep modes and efficient scheduling to conserve energy (Section 2.3), an IoT OS may aim to provide the means to allocate, limit, and isolate the energy consumption of certain software components. Through its small trust base (F1), an IoT OS may become capable of allocating and enforcing policies over energy usage (given a known/estimated total of available energy) [2] over distinct software components (e.g. user applications and kernel drivers). This may help prolong the life of an IoT device [128], guarantee components do not exceed their allocated energy resources, and ensure other components cannot steal energy reserved for others (paralleling our discussion on memory isolation).

The allocation, isolation, and mediation of access to important resources, such as memory and energy, serve as critical steps towards the enforcement of an IoT OSs established trust models (Section 6.2). An IoT OS must proactively utilize hardware- or software-based security foundations and employ any additional mechanisms to strengthen and deliver the properties promised by said models.

Comparing RIOT to Tock on F3 Alluded to by its lack of established trust models and made evident through our experimental work, RIOT currently does not

employ resource isolation. Although RIOT is modular-by-design, once components are compiled together, they run with the same privileges and have full access to the single physical address space accessible to all user applications and the kernel. Despite this, we learned that RIOT makes use of PMP on ARM Cortex-M devices containing an MPU. However, we also learned that multiple privilege modes are never used, even on platforms with support for such. Privilege separation has relevance to security when used in tandem with PMP (Section 5.5), but lack of privilege separation does not completely devalue use of PMP, e.g., for stack protections, rather, it means every component runs with the same privilege and may freely disable PMP, in turn removing any configured memory protections and granting unrestricted access to any memory location. Executing code in this manner certainly raises concern for security, particularly due to the fact that RIOT and its applications are/will primarily be written in the memory-unsafe C language. Due to the foundational nature of PMP and privilege modes, enforcing privilege separation in RIOT would be highly non-trivial. For example, if the kernel were to be run as privileged and user applications as unprivileged, a standard system call interface would need to be made, then, every user application and kernel interface (including kernel drivers and libraries) would require changes to support the system call interface. Moreover, these changes sadly would not apply across the wide variety of hardware supported by RIOT, as some completely lack underlying hardware support.

As alluded to in previous sections, Tock facilitates and employs (via its trusted kernel core) both hardware- and software-based memory isolation techniques, isolating untrusted kernel components through language-based sandboxing and user processes

through hardware-based privileges modes and PMP. Within the kernel, Tock leverages Rust's type-safe and memory-safe features to separate trusted and untrusted components. The untrusted capsule components (written in Rust) may not use the `unsafe` keyword, resulting in compile-time sandboxing, such that by design, capsules, for example, cannot dereference arbitrary pointers. This both stops arbitrary memory access to kernel memory, and any access to the memory of a given process that is making use of a capsule. Outside the kernel, Tock uses of hardware-based privilege modes and PMP enables the enforcement of memory isolation between each user application and the kernel. Configuration is done by the Tock kernel, ensuring memory allocated to each process is the only accessible memory to each respective process. To our knowledge, Tock does not provide any means to allocate or enforcibly limit energy consumption over components. As a result, devices constrained in energy may become subject to resource exhaustion attacks whereupon exploited or malicious untrusted components constantly execute until total device availability is compromised due to a lack of finite energy.

6.4. Access Control Mechanisms (F4)

Facilitate the explicit establishment and maintenance of controlled/mediated exclusive communication channel(s) for/between kernel components and user applications

An inextricable challenge arises when device resources are exclusively allocated and isolated to individual software components (F3). That is, imposed isolation boundaries prevent components from directly communicating with or accessing resources

owned by others. Thus, developing *access control mechanisms* allowing for inter-component communication and resource sharing in a controlled/mediated fashion becomes necessary, but may require more complex designs [7] to ensure proper adherence to important security principles. For example, implementation of access control mechanisms, such as system call interfaces to access kernel services or kernel-mediated shared memory, should avoid over-allocation of resources, aim to only extend privileges when needed, in a controlled manner and for the shortest duration necessary, and deny any access not explicitly established by-default [140, p.21-22][74]. Further, the consumption of finite resources, such as energy, should be traced back to the consumer (e.g. user application) and counted towards it total energy quota depending on the enforced policy [2]. In turn, this may allow an OS to maintain resource isolation and provide access controls in a *fine-grained* manner.

Notably, access controls must follow after techniques used to achieve resource isolation (F3) but similarly may be provided through software or hardware. Interestingly, academic work proposing augmentation [206, 115] or replacement of [211] hardware-based memory protections on (constrained) low-end devices has made it evident that existing (Section 4.1.1) COTS hardware-based access control mechanisms are inherently coarse-grained. The limited number of hardware-based privilege levels and PMP regions are limitations that may incline developers towards developing software-based access controls in tandem with or parallel to hardware-based mechanisms. Software may provide greater flexibility and fine-grained control [129, 42, 183], but, for example, may incur runtime costs [55, 52, 120] and/or rely on compile-time guarantees [129, 42] which may not provide the same isolation guarantees of

hardware-based isolation, particularly in low-level systems, such as OSs, that may fundamentally require some use of code that cannot be guaranteed completely safe at compile time (Section 4.2.2). Similar arguments apply with respect to energy management [128]. Thus, an IoT OS may better serve by making use of available hardware-based mechanisms and complementing or compensating for any hardware limitations through software-based means.

Comparing RIOT to Tock on F4 Given every software component comprising RIOT is compiled into a singular monolithic binary, it becomes straightforward for any component to directly invoke services (in the form of function calls) provided by other components. RIOT's limited use of PMP precludes the need for complex hardware-enforced system call interfaces (backed by privilege modes). Further, given RIOT is developed in C, it can not and does not gain compile-time guarantees provided by memory-safe alternatives. Notably, this results in RIOT lacking adequate resource isolation (particularly memory isolation), preventing RIOT from providing any tamperproof means of enforcing and providing meditation through access control mechanisms over components.

In Tock, PMP-isolated processes can safely and explicitly communicate with one another using kernel-provided IPC mechanisms. Alternatively, for a process to communicate with the kernel (including Tock capsules), Tock provides a minimal set of system calls (seven in total [196]) enforced as the only entry points via privilege modes, which in turn enable user space processes to communicate directly with the kernel in a safe and controlled manner, while only incurring performance penalties

for context switches and subsequent dynamic PMP configuration. In the kernel, Tock capsules are sandboxed through Rust's compile-time memory safety guarantees. Rust functions and traits are only accessible to others through explicit use of the `pub` access modifier (Section 4.2.2. As a result, capsules benefit from fine-grained access controls. Tock capsules may only invoke other publicly defined kernel components they have been linked with and are restricted from directly accessing user processes due to restrictions over the `unsafe` keyword (enforced at compile-time).

6.5. Fault Isolation and Recovery (F5)

Ensure component faults do not cause other components to fault and, if possible, build in mechanisms to recover faulted components

In general, a *fault* may be defined as any disruptive error induced either by software (e.g., code panics) or hardware (e.g., CPU exceptions, corrupted memory, etc), any of which may result in a software component or, potentially, an entire system crashing. *Fault isolation* ensures faults induced by or directly impacting one, already isolated (F3), component do not cause any other (potentially interdependent) component(s) to fault/fail as well [104, 34]. Differing from *resource* isolation, *fault* isolation refers to the additional steps necessary to ensure all resources used solely by a faulted component are safely deallocated, allocated resources shared between

components are preserved (e.g., mitigating problems arising from *state spill*[1]), and all future invocations to services formerly provided by the, now likely terminated, faulted component do not block or halt execution of the caller–instead, invocations may, for example, result in meaningful error messages callers can safely handle [129].

Fault isolation serves as the prerequisite step towards allowing subsequent recovery of faulted components. In its simplest form, *fault recovery* allocates resources back to a failed component then restarts it[2]. Overall, fault isolation and recovery directly work towards availability of select components (by reducing the amount of damage a fault may cause and maintaining access to component functionality and services upon recovery), a security goal that may be particularly important for IoT systems typically deployed in (potentially difficult to access) remote locations and/or large (potentially hard to manage) quantities.

An OS aiming to adequately satisfy this criterion should make efforts to ensure fault isolation and recovery is enforced over every untrusted component. In our proposed threat model (Chapter 3), untrusted components are those which may be provided by untrusted entities, such as user applications and kernel components residing outside of the trusted core kernel, thus, to work towards security goals such

[1]State spill [43] occurs when interaction between software components leads to lasting state changes, such that future correctness depends on resulting changed states. For example, in a typical generic client-server example, the server may hold state information representing its clients' progress. This may represent state spill, where if the server crashes and loses stored state information, clients will likely also fail [129].

[2]Systems going beyond this simple form of fault recovery may attempt to maintain state information then restore the faulty component back to its prior functional state, such that it can continue processing requests made before or during the fault [185].

as availability, fault isolation and recovery would ideally be guaranteed for any such components. More generally, as trust may be allocated across components differently from system to system, we believe OSs should ideally use priorly established trust models (F2) to guide and document guarantees (or lack thereof) relating to this criterion.

Comparing RIOT to Tock on F5 As discussed, RIOT lacks a prerequisite for fault isolation and recovery—resource isolation. Utilized software components are compiled into one monolithic binary, making each component indistinguishable from one another which, when loaded for execution, share the same global stack and heap for data storage. As a result, when a fault, such as an invalid memory access, occurs in RIOT, the only viable outcome remains to be that which we continually observed in our experiments: a system-wide reset[3]. All in all, these aspects of RIOT directly prevent it from providing fault isolation (or recovery) at the component level, impacting its ability to maintain the availability of components unrelated to that which just triggered a fault.

Next, Tock provides fault isolation for and recovery of user applications, but not for anything below that at the kernel level (such as its untrusted capsules). If we recall,

[3]One might think that because RIOT threads are managed/scheduled, a fault originating from one of these threads should be traceable to said thread. This is not the case, even if execution was within a thread prior to the fault, a system reset likely remains as the only option. Returning execution back to the point before the fault occurred would immediately result in the same fault again, and because code exists as one amalgamated unit in RIOT, it can not be simply determined where code pointing to the beginning of a thread exists; execution must be trapped to an exception handler which would either reset the system itself or wait until a hardware event (i.e., watchdog timer [135, p.409-414]) triggers a reset [19, p.611-615].

user applications are flashed into distinct regions of Flash memory, granted a predefined exclusive region in RAM for data storage, then isolated through hardware-based means (PMP). Further, any attempts at sharing memory with other user applications must go through the kernel, who maintains ownership and manages said memory; eliminating problems of state spill, as shared resources are maintained even when either of the involved applications fault. If a fault were to occur, execution is trapped to privileged kernel fault handling code (in turn terminating execution of the faulted user application), then, because each application is established with exclusive memory regions, the kernel fault handler has the ability to make informed decisions, e.g., on whether the process should be restarted/recovered or remain unscheduled/terminated. Moving into the kernel, we last discussed how Tock capsules are adequately isolated through software-based sandboxing via forbidden use of unsafe Rust code. However, this does not guarantee fault isolation or recovery. Capsules are compiled together as part of the kernel, must invoke encapsulated unsafe code provided by the core kernel to access underlying hardware peripherals (c.f. Section 5.2), and as a consequence may share references to the same memory (potentially leading to problems related to state spill). Together, these characteristics make capsules indistinguishable from any other kernel code upon execution and prevent fault isolation and recovery.

6.6. Cryptographic Primitives and Libraries (S1)

Provide time-tested and usable cryptographic algorithms that satisfy a range of user requirements and application needs

IoT OSs utilizing foundations and working towards satisfying the aforementioned criteria may also find the inclusion of cryptographic primitives and libraries an essential building block relating to security. OS kernel components and user applications may require cryptographic primitives for security critical functionality such as network authentication and communication (S2), resource access authorization, object/data security, key management [201], and secure firmware updates (S3). Importantly, the secrets (e.g., *cryptographic keys*) underpinning such functionality may highlight the importance of a properly isolated trust base (F1), even more so for multi-tenant systems. As defined, the OS's trust base may be the only software fully trusted to securely encapsulate and maintain an interface to secret material even when coexisting with other untrusted or malicious components.

Additionally, a novel challenge arises when dealing with low-end IoT devices. That is, attacks may be initiated by adversaries with much greater computational power than an IoT device [14]. As a result, cryptographic implementations on these constrained low-end devices require extra care to meet specifications and avoid bugs that can lead to information leakage [201], Thus, it may be particularly beneficial for IoT OSs to rely on time-tested [140, p.22] cryptography libraries and safe hardware implementations [97, 5] that have been thoroughly audited, rather than newly invented or independently implemented. Further, as cryptographic libraries are typically exposed through APIs, in which fallible developers may call, libraries should ideally be designed or chosen such that they produce the least amount of unexpected surprises [140, p.22-23] to avoid potential software vulnerabilities caused by accidental misuse [180, 68]. Finally, to motivate their use in system components and

user applications, algorithms should have only minor impact on system memory and power budgets, such that security benefits outweigh any costs [140, p.23].

Comparing RIOT to Tock on S1 RIOT provides internally developed implementations of several cryptographic algorithms within its system libraries and integrates several external cryptography libraries written in C, such as TweetNaCl [38], micro-ecc [112], the High Assurance Cryptographic Library (HACL) [222], and the RELIC toolkit [12]. This gives RIOT users a varied set of options, some implementations widely-adopted, efficiently and safely implemented, or time-tested. This allows RIOT to meet a broad range of user requirements and application needs.

Currently, Tock only officially supports hardware-backed cryptography explicitly provided by the platforms they support such as AES encryption, hash-based message authentication codes (HMACs), and random number generators (RNGs). The Tock kernel does not have any additional support for software-based algorithms or asymmetric cryptography, leaving a gap in cryptographic primitives and libraries, compared to RIOT. Tock's strict rule to deny `unsafe` Rust code in their kernel capsules, precludes adopting popular Rust-based cryptography libraries [169] that use `unsafe` code in their implementations. This may hinder Tock's ability to provide its users with formally audited and time-tested cryptography, generally available to the wider Rust community. It could also motivate Tock developers to write their own cryptographic algorithms, which typically requires considerable developmental efforts [201] and might lead to vulnerable implementations, if sufficient care is not taken. However, user applications are allowed to import and use available cryptog-

raphy libraries within their implementations; since these imported libraries will be part of the resulting user process, they will consequently be isolated from the rest of the system by the Tock kernel.

6.7. Secure Communication Protocols (S2)

Implement and maintain reliable network protocol stacks to support the establishment and maintenance of secure communication channels, following open protocol standards where applicable

A defining characteristic of an IoT device is its ability to communicate data over some medium, with an emphasis on wireless technologies, to other similar devices or gateways. As outlined in our threat model (Chapter 3), such communication is susceptible to a wide variety of attacks [149] and require careful consideration to avoid or mitigate, particularly since these devices will often not be deployed in a closed and trusted network [14]. Open protocol standards, such as those developed by the IETF, typically have higher quality and interoperability due to broader expert scrutiny and a greater number of carefully considered revisions which may result in better security [201]. Moreover, securing network communications typically requires integration with cryptographic primitives and protocols [152, 102]. Thus, in general, an IoT OS may better serve when networking stack implementations (e.g., those which include network, transport, and application layer protocols) are based upon open protocol standards, integrate with standard cryptographic primitives and libraries (S1) to allow for establishment and maintenance of secure network commu-

nications, and additionally made developer-friendly [201] such that exposed APIs are hard to misuse.

We may take *Datagram Transport Layer Security* (DTLS) [152] as one example protocol with relevance to securing communication in the IoT. In contrast to *Transport Layer Security* (TLS) [151], a cryptographic protocol (and standard) for establishing and maintaining secure communication over TCP, DTLS provides equivalent security for communication over UDP. DTLS allows for secure communication with less overhead compared to protocols ubiquitous in the IoC, such as TLS. Yet similarities to TLS, such as its open standardized nature, reliance on well-established cryptographic primitives, and overall specification make DTLS an enticing and potentially important option to be integrated with IoT OS networking stacks.

Comparing RIOT to Tock on S2 RIOT supports several networking stacks and protocol implementations, focusing primarily on open standard protocols [70, 85] relevant to the IoT. RIOT's default stack, the Generic (GNRC) [103] network stack, provides a fully-featured IPv6 over IEEE 802.15.4 implementation via the 6LoWPAN adaption layer [123] that provides extensions to the UDP, RPL [209], and TCP transport layers. Application layer protocols can integrate with GNRC, for example, to exchange CoAP requests over UDP. Additionally, development towards secure communication in RIOT is exemplified by the integration of the DTLS with GNRC [85], an important step for allowing efficient secure end-to-end communication for RIOT-based devices. Beyond this, RIOT makes available vast open-source libraries [154] providing alternative networking stacks (e.g., NimBLE [10], OpenThread [67], etc).

Later discussed, RIOT's library support for both networking and cryptography can be combined to satisfy secure communications and consequently support security-critical networking tasks such as remote firmware updates (S3).

Similarly, Tock includes (or has plans for [191]) its own set of networking stacks that support UDP, IPv6, 6LoWPAN, IEEE 802.15.4, and BLE [116]. Recalling that Tock is written in Rust and has strict requirements on capsule implementations, Tock consequently does not currently integrate with any additional third-party libraries. This evidently adds extra developmental costs, leading to lack of support for a wider variety of protocols (such as TCP and application layer protocols) within Tock, but may allow Tock developers to implement these protocols over time in ways that fully benefit from the memory safety guarantees provided by Rust. To our knowledge, Tock currently has little explicit integration of cryptographic protocols with its networking stacks and, for example, does not support secure end-to-end communication (e.g., DTLS).

6.8. Secure Remote Firmware Updates (S3)

Provide mechanisms to remotely receive authenticated and integrity protected firmware images, allowing already running firmware to be replaced/updated at runtime

The firmware on any given IoT system may contain exploitable security flaws, some of which may be patched through firmware updates. IoT devices however, may be deployed remotely, in inaccessible environments, or in manually unmanageable quan-

tities. Thus, to enable users to securely provision firmware updates for their devices it may be paramount for an IoT OS to support remote firmware updates. Ideally, the firmware *image* provided over an RF communication channel, i.e., *over-the-air* (OTA), to a device should be integrity protected and authenticated to prevent unwarranted/malicious modifications when in-transit and allow devices to validate the image and its sender upon reception [124], which, as we discuss below, pose challenges needing careful considerations with respect to low-end IoT devices.

For remote firmware updates, an IoT OS must have the ability to remotely retrieve firmware images and/or manifests containing additional meta-data [125] from an update server, then parse and verify the manifests, and finally store the firmware image onto Flash memory [217]. We reiterate that an OS needs a trust base (F1) to act as the (or extend trust to another) component that can securely determine the validity of received firmware then load or store it into any available slot [217, 108]; without this properly defined and isolated trusted component, the update process may be overly exposed to potential tampering by attackers (Section 3.2).

Consequently, cryptographic algorithms (S1) may be required to verify manifests and protect against modification [124], yet such a prerequisite may demand considerable care to mitigate against potential attacks. Part of the work presented by Ronen et al. [167] involved the recovery of symmetric cryptographic keys used to encrypt and authenticate firmware images of popular Philips Hue smart lamp products powered by wirelessly communicating low-end devices. Critically, the recovered symmetric keys were found to be reused globally for all devices of the same type, which

allowed the researchers to remotely update these devices with their own malicious firmware. As the authors conclude, reuse of symmetric keys between devices poses a big security risk. Thus, research into and support of potential viable alternatives, such as asymmetric cryptographic primitives and any supporting infrastructure, may be necessary to mitigate against certain problems, such as firmware authenticity and key reuse.

Additionally, networking capabilities (S2) providing secure end-to-end communication of any firmware/manifest data may also be necessary, since obtaining firmware images is likely the first step an attacker may take to gain valuable insights on device-specific configuration that can lead to system exploitation [124].

Finally, for scrutinized mechanisms that provide better interoperability [201], an OS may benefit from adhering to open secure firmware update standards (e.g., IETF standards [124, 125]).

Comparing RIOT to Tock on S3 RIOT has experimental support for secure OTA firmware updates. As outlined by the work of Zandberg et al. [217], this support follows the Software Updates for Internet of Things (SUIT) IETF working group specifications [124, 125]. Consequently, the implementation in RIOT following said SUIT specifications has now been integrated into the mainline RIOT code as an experimental feature [162]. RIOT integrates a necessary bootloader [160] with its networking, cryptographic, and other relevant supporting libraries to adhere to these open standards, facilitate communication with required infrastructure, and achieve

this initial support for secure OTA firmware updates.

Tock's kernel acts as a bootloader by default, with the ability to load application binaries following its custom Tock Binary Format (TBF) [193], however, Tock currently does not itself support secure remote firmware updates. From earlier sections, it is evident that Tock has many of the features needed for this, but further developmental efforts are needed to add supporting functionalities, cryptographic primitives, and communication protocols. This may first require additional considerations and discussions on whether Tock should/can adhere to open standards, such as those proposed by the SUIT working group.

6.9. Source Code Auditability (S4)

Maintain source code and supporting infrastructure in a manner that is conducive to auditing processes, making clear any security critical code/components

This final proposed (supporting) criterion recommends IoT OSs be developed and supported in a manner conducive to auditing processes. Generally speaking, the auditability of any given system is established through a continuous cycle of distinct processes that take into account the organization and the resources it is managing (those of which an involved party might directly rely upon), resulting in the system becoming knowable not only to the maintaining organization, but also to any relevant third-party agencies or users/stakeholders [204]. This may require use of a normative framework allowing involved parties to evaluate any referential infrastructure (e.g.,

code repositories, continuous integration services, etc) used by an organization to maintain the IoT system, and verify its accountability, reliability, and trustworthiness [205, 204, 70].

Thus, to satisfy this requirement, the source code and implementation of any IoT OS should be available and supported by a variety of techniques to ease the process of formal or informal auditing, particularly for components critical to overall system security. These techniques can include [70, 31]: providing code in a fully open-source manner for increased exposure and probability of finding bugs, including unit/integration tests, keeping security critical components small (F1), clearly marking potentially hazardous use cases, and providing thorough documentation all of which may improve the auditability and security of a system [76].

Comparing RIOT to Tock on S4 RIOT is openly developed primarily under the GNU Lesser General Public License (LGPL), containing some external sources published under separate licenses. RIOT's project repository [158] contains documentation, tutorials, code examples, unit testing, continuous integration testing performed by web-based services, benchmarking tests, issue tracking, and detailed release notes. Additionally, external documentation is automatically created from inline comments of the source code itself, using a tool call Doxygen[4]. The highly modular fashion of RIOT allows potential auditors to easily distinguish each RIOT component, however as mentioned earlier, RIOT's lack of established and documented trust models may leave it unclear how each integrated component can or will interact with another and

[4] https://www.doxygen.nl

whether if access to the greater shared physical memory address space is mediated in any way upon execution. Despite this, the ample publicly available documentation and testing can aid in code audits of individual components, but still may leave a challenging task when dealing with larger systems integrating many components together, any of which could carry vulnerabilities potentially compromising the entire system, exacerbated by the lack of a small trust base (F1) and related resource isolation (F3) in RIOT.

Tock is openly developed and can be licensed under either the Apache or MIT permissive licenses. Tock's project repository [198] contains extensive documentation, tutorials, increasing amounts of platform-specific unit testing, issue tracking, detailed release notes, and historical records of several focused working groups consisting of developers openly discussing and developing particular aspects of Tock. Together, these should in theory allow any third-party to learn and gain necessary knowledge about Tock, facilitating understanding and providing context when attempting to audit any code. Finally, because Tock is implemented in Rust, a large majority of its code is believed to adhere to the memory safety features provided by Rust, presumably allowing an entity attempting to audit Tock to focus on the trusted code explcitely marked with the `unsafe` keyword, and checking untrusted components for cases that extend beyond Rust's memory safety promises.

6.10. Tabulated Summary and Discussion

In this section, we briefly summarize our findings, discuss insights, and offer opinions on our selected IoT OSs with respect to our established foundational and supporting criteria.

Criterion		RIOT	Tock
F1.	Small Trust Base	○	●
F2.	Established Trust Models	○	●
F3.	Resource Allocation and Isolation	◑	◑
F4.	Access Control Mechanisms	○	●
F5.	Fault Isolation and Recovery	○	◑
S1.	Cryptographic Primitives and Libraries	●	◑
S2.	Secure Communication Protocols	●	○
S3.	Secure Remote Firmware Updates	●	○
S4.	Source Code Auditability	●	●

Table 6.1.: Comparing criteria satisfied by selected platforms

● adequately satisfied; ◑ partially but inadequately satisfied; ○ unsatisfied

Referring to Table 6.1 and starting with RIOT, we note each of our five foundational criterion is either inadequately or wholly unsatisfied by RIOT. Knowingly, RIOT supports a wide range of devices, including those based on MCUs physically lacking hardware-based security foundations (Section 4.1), such as multiple privileges modes and PMP. As a consequence, satisfying these criteria purely through hardware-based mechanisms uniformly across its supported platforms is not possible. However, RIOT does support many devices containing hardware-based foundations, such as the COTS ARM-based device we used in our experiments. As discussed in Section 5.6, RIOT does in fact configure and utilize PMP when available, but,

for example, does not use hardware privilege modes available on the very same devices. Thus, although PMP may be used to prevent certain access violations (on configured regions of memory), lack of privilege modes or alternative foundational software-based mechanisms precludes adequate resource isolation in RIOT. This fact was pronounced when components in RIOT, including those which we idealistically modeled as untrusted in our general threat model, such as user applications, were found able to disable PMP altogether (Section 5.5).

Consistent with our definition of a foundation, supporting privilege modes in RIOT would require significant code changes (to introduce and define standard system call interfaces), all of which would be incompatible with devices lacking such hardware. Further, RIOT is written in C and thus unavoidably remains vulnerable to common memory corruption errors, some of which may result in access violations if mistakenly triggered or actively exploited. Even though RIOT is openly developed and supported by procedures and infrastructure which may help catch some of these errors, history has made evident that development of systems written in C with similar measures, such as Linux, have not remained error free. Continuing with Linux as a relatable example, there has been noticeable interest, research, and efforts into incrementally introducing memory-safe languages, such as Rust, for kernel development [106, 137]. Similarities between RIOT and Linux may make consideration of similar incremental efforts worth investigating, particularly if it can help prevent or reduce introduction of memory corruption errors.

To summarize RIOT, we note how it adequately satisfies the criteria we hold as

supportive to our outlined security goals, but not those posited as foundational, for example, establishment of a small trust base or distinct adequately isolated components through use of security foundations. It is due to the combination of hardware constraints, or framed more positively – broad support, and software design that we conclude RIOT inadequately supports such foundational criteria across all of its supported devices. As a final note, we believe RIOT and its users could benefit from an established set of clearly documented trust models (F2) detailing any security guarantees (or lack thereof) and differences across RIOT's varying supported hardware. This may provide RIOT users with a more clear and explicit reference to what they should expect when relying on RIOT, help determine any trade-offs between different supported devices, and decide whether RIOT is right for their specific IoT use case.

Pressing on, we next discuss Tock and how it contrasts with RIOT. Through notable use of and reliance on hardware- and software-based foundations, Tock's core kernel is firstly isolated from other kernel components through enforced compile-time memory safety guarantees, then, subsequently isolated from user applications upon execution through combined use of hardware-based PMP and privilege separation. Due to reliance on hardware-based mechanisms, Tock can only officially support devices with hardware support for both multiple privilege modes and PMP. As a result however, these same mechanisms adequately allow Tock's core kernel to act as the entire OSs trust base, and enforce adequate memory isolation between the core, untrusted kernel components, and user applications. Albeit, such mechanisms do not necessarily prevent exploitation of individual components altogether, as experi-

mentation showed (Section 5.3), but in certain cases may make it significantly less likely/more difficult for an infected component to affect other distinct parts of the system by eliminating potential to access resources beyond what has been explicitly allocated. This holds particularly true for user applications in Tock, as discussed, hardware-based mechanisms employed by Tock additionally allow for fault isolation and recovery, e.g., those which may be caused by memory access violations, for user applications. However, Tock kernel components run with the same privilege as the core kernel (small trust base) itself and do not have any built-in software-based [129, 42] fault isolation or recovery mechanisms, thus, faults caused by these purely software sandboxed components are not isolated (i.e., may require/cause a system-wide reset). Moreover, Tock does not provide a means to enforce allocation and isolation of energy to specific components, leaving resource exhaustion attacks as a possibility. Relevantly, these important differences along with additional security-specific guarantees, assumptions, and expected involved entities are publicly documented, which together adequately establish Tock's trust model(s).

Finally, Tock's Rust-based implementation, paired with strict rules imposed on the implementation of untrusted kernel components (i.e., capsules), leaves implementation of functionality, such as asymmetric cryptography potentially required for secure communication and firmware updates, solely to Tock contributors, potentially isolating Tock from the greater Rust community. The security benefits of excluding unsafe code may outweigh this temporary loss of functionality, if assuming additional developmental work will allow Tock to eventually adequately satisfy supporting criteria. In the meantime, users may directly import third-party libraries into user

applications (e.g., those written in C or Rust) and rest assured that potential errors in these libraries will face the same isolation boundaries imposed over the entire application and that applications may be safely recovered (e.g., restarted) if such errors cause a fault.

Overall, our proposed foundational and supporting criteria aim to broadly capture tangible considerations for securing IoT OSs supporting low-end devices. Evidently, these criteria together aim to address our specific threat model and take into account potential growing complexity within the IoT, such as potential need for multi-tenant OSs supporting low-end devices. Ultimately, this means such criteria may not entirely capture every IoT use case. Simple uses cases may only demand a subset of these criteria to be deemed secure, other use cases may require specific considerations particular to said use case. However we believe, in general, and particularly for OSs positioning themselves as broadly applicable to any IoT use case [33], proposed criteria may help developers make informed decisions relating to security and, once implemented and further scrutinized, make way for time-tested formal principles to develop and predicate the design of secure IoT OSs.

Chapter 7.

Closing Remarks

The primary objective of this thesis was to establish a set of criteria operating systems supporting low-end devices in the Internet of Things may satisfy in order to define, establish, and more broadly maintain security. To approach this objective, we delved into hardware- and software-based security to identify important mechanisms—security foundations—available for use on low-end devices. We then played the role of attacker and set out to examine these foundations through experimentation on two open-source IoT OSs with academic origins, RIOT and Tock. Through this iterative process, we gained insight into the utilization of security foundations and, more importantly, the role and limitations of these foundations in preventing or limiting damage cause by device exploitation. Finally, by combining gained knowledge with additional research into the literature we proposed a set of nine criteria, four towards realizing an initial secure environment for the execution of software components comprising IoT OSs followed by another four towards

maintaining and supporting security of said systems; we compare RIOT and Tock against these criteria which, when combined with our experiments on these very OSs, provided a detailed view into the security guarantees provided by each system and the foundations relied upon to achieve any such guarantees.

7.1. Future Work

RIOT and Tock stood as two exemplary IoT OSs with strong academic origins and active open-source communities, but they are not the only IoT OSs around. Expanding this work to cover several more IoT OSs, including those without academic origins, such as Zephyr OS [218] or Drone OS [202], could lead to additional insights. Moreover, our work focused primarily on OS support for Class 1 and Class 2 devices; we believe including systems targeting Class 0 devices would also be interesting. Specifically, FreeRTOS [6] (written in C) and RTIC [168] (written in Rust) may contrast well with our coverage of RIOT and Tock respectively. However, the frameworks/OSs that target Class 0 devices may have different characteristics and requirements, with a bigger focus on specialized singular applications that have multiple related concurrent tasks (i.e., interrupt-triggered events), and thus may require only a subset of our proposed criteria or different criteria altogether.

Finally, our discussions on memory-safe programming languages coupled with the surge of research and interest in this area makes us believe memory-safety may play an important role in the development of any low-level system. Tock is not the only system benefiting from a memory-safe language, there are additional systems writ-

ten in Rust [129, 42, 150, 202] and other memory-safe languages [52, 80] which may be interesting targets to explore operating system security more generally. In many scenarios, hardware is expected to be required to enforce security (and trust) guarantees, however, as we learned, memory-safe languages may provide guarantees of their own, at compile-time. Exploring these guarantees and assessing the advantages, disadvantages, and limitations in comparison to hardware and traditional systems written in C may be worthwhile.

7.2. Conclusion

Designing secure systems is a notoriously difficult task, in which each design choice affects numerous parts of the system, including its ability to provide security. Heuristics, criteria, and design principles all try to encapsulate the nuances of security and help better guide developers towards creating secure systems. Before these systems get created or are further developed, it is important to continually question, address, and update these methods and solutions. Then, focusing on developing consolidated platforms, such as operating systems, with solid foundations may allow the community to try and get security right as best we can, which in turn can provide new insights into additional criteria towards securing these systems and the applications they serve.

Notably, the IoT is becoming ubiquitous and these Internet communicating devices have the ability to interact with the physical world, seamlessly integrate into our environments, and potentially affect individual safety and privacy. Thus, carefully

developing reliable systems that rely on strong security foundations, integrate time-tested libraries, make use of trusted standards and specifications, and adhere to security best practices are all likely to improve overall system security and prevent unauthorized use. In the Internet of Things, as well as more generally, the main idea of designing operating systems to satisfy security criteria from the start, is to ensure adequate considerations towards the nuances and avoid situations where reliant applications and users become unable to compensate for lack of underlying support.

References

[1] Tigist Abera, N. Asokan, Lucas Davi, Farinaz Koushanfar, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. "Invited - Things, Trouble, Trust: On Building Trust in IoT Systems". In: DAC. Association for Computing Machinery, 2016. DOI: 10.1145/2897937.2905020.

[2] Joshua Adkins, Bradford Campbell, Branden Ghena, Neal Jackson, Pat Pannuto, and Prabal Dutta. "Energy Isolation Required for Multi-tenant Energy Harvesting Platforms". In: *Energy Harvesting and Energy-Neutral Sensing Systems, ENSsys@SenSys*. ACM, 2017, pp. 27–30. DOI: 10.1145/3142992.3142995.

[3] Joshua Adkins, Branden Ghena, Neal Jackson, Pat Pannuto, Samuel Rohrer, Bradford Campbell, and Prabal Dutta. "The signpost platform for city-scale sensing". In: *Information Processing in Sensor Networks, IPSN*. IEEE, 2018, pp. 188–199. DOI: 10.1109/IPSN.2018.00047.

[4] Kofi Sarpong Adu-Manu, Nadir H. Adam, Cristiano Tapparello, Hoda Ayatollahi, and Wendi B. Heinzelman. "Energy-Harvesting Wireless Sensor Net-

works (EH-WSNs): A Review". In: *ACM Trans. Sens. Networks* 14.2 (2018), 10:1–10:50. DOI: 10.1145/3183338.

[5] Ehsan Aerabi, Milad Bohlouli, Mohammad Hasan Ahmadi Livany, Mahdi Fazeli, Athanasios Papadimitriou, and David Hély. "Design Space Exploration for Ultra-Low-Energy and Secure IoT MCUs". In: *ACM Trans. Embed. Comput. Syst.* 19.3 (2020), 19:1–19:34. DOI: 10.1145/3384446.

[6] Amazon Web Services. *FreeRTOS*. URL: https://freertos.org/.

[7] Amazon Web Services. *FreeRTOS-MPU*. URL: https://freertos.org/FreeRTOS-MPU-memory-protection-unit.html.

[8] Nadav Amit, Amy Tai, and Michael Wei. "Don't Shoot down TLB Shootdowns!" In: EuroSys '20. Heraklion, Greece: ACM, 2020. DOI: 10.1145/3342195.3387518.

[9] M. Antonakakis and 18 others. "Understanding the Mirai Botnet". In: *USENIX Security 2017*, pp. 1093–1110.

[10] Apache Mynewt Project. *NimBLE*. URL: https://github.com/apache/mynewt-nimble.

[11] Apple. *Apple Watch battery*. URL: https://www.apple.com/watch/battery/.

[12] D. F. Aranha, C. P. L. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao. *RELIC is an Efficient LIbrary for Cryptography*. https://github.com/relic-toolkit/relic.

[13] Emekcan Aras. "Security and Reliability for Emerging IoT Networks". PhD thesis. KU Leuven, Feb. 2021.

[14] Jari Arkko. *Keynote: Call for Action: The Internet Threat Model Needs a Change*. RIOT Summit 2019. URL: https://www.youtube.com/watch?v=JX6gzCeU7H0&list=PLDXXQJiSjPKGZPirt2f7-6LrCtcnAoC0i&index=2.

[15] *AMBA 3 APB Protocol Specification*. Version IHI 0024B. ARM. Aug. 2004.

[16] *AMBA 3 AHB-Lite Protocol Specification*. Version IHI 0033A. ARM. June 2006.

[17] *CoreSight Components Technical Reference Manual*. Version DDI 0314H. ARM. July 2009.

[18] *Arm Cortex-M3 Technical Reference Manual*. Version Revision r2p0. ARM. Feb. 2010.

[19] *ARMv7-M Architecture Reference Manual*. Version DDI 0403E.b. ARM. Dec. 2014.

[20] *ARMv8-M Architecture Reference Manual*. Version DDI 0553A.e. ARM. June 2017.

[21] *ARMv6-M Architecture Reference Manual*. Version DDI 0419E. ARM. June 2018.

[22] *Arm Cortex-M4 Technical Reference Manual*. Version Revision r0p1. ARM. May 2020.

[23] *STM32L100xx, STM32L151xx, STM32L152xx and STM32L162xx advanced Arm-based 32-bit MCUs*. Version RM0038. ARM. Jan. 2020.

[24] *STM32WLEx Reference manual*. Version RM0461. ARM. Nov. 2020.

[25] ARM. *TrustZone for Cortex-M*. URL: https://www.arm.com/why-arm/technologies/trustzone-for-cortex-m.

[26] Arm. *CMSIS*. URL: https://developer.arm.com/tools-and-software/embedded/cmsis.

[27] Arm. *MPU programmers' model changes for the ARMv8-M architecture*. URL: https://developer.arm.com/documentation/100699/0100/Introduction/MPU-programmers--model-changes-for-the-ARMv8-M-architecture.

[28] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, Aug. 2018.

[29] Matt Asay. *Why AWS loves Rust, and how we'd like to help*. AWS Open Source Blog. URL: https://aws.amazon.com/blogs/opensource/why-aws-loves-rust-and-how-wed-like-to-help/.

[30] Hudson Ayers, Paul Crews, Hubert Teo, Conor McAvity, Amit Levy, and Philip Levis. "Design Considerations for Low Power Internet Protocols". In: *Distributed Computing in Sensor Systems, DCOSS*. IEEE, 2020, pp. 103–111. DOI: 10.1109/DCOSS49796.2020.00027.

[31] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch. "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT". In: *IEEE Internet of Things Journal* 5.6 (2018), pp. 4428–4440. DOI: 10.1109/JIOT.2018.2815038.

[32] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt. "RIOT OS: Towards an OS for the Internet of Things". In: *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 2013, pp. 79–80. DOI: 10.1109/INFCOMW.2013.6970748.

[33] Emmanuel Baccelli, Oliver Hahm, Matthias Wählisch, Mesut Günes, and Thomas Schmidt. *RIOT: One OS to Rule Them All in the IoT*. Tech. rep. hal-00768685v3. INRIA, 2012. URL: https://hal.inria.fr/hal-00768685/file/RR-8176.pdf.

[34] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamaric, and Leonid Ryzhyk. "System Programming in Rust: Beyond Safety". In: *ACM SIGOPS Oper. Syst. Rev.* 51.1 (2017), pp. 94–99. DOI: 10.1145/3139645.3139660.

[35] BananaMafia. *Brute-Forcing x86 Stack Canaries*. URL: https://bananamafia.dev/post/binary-canary-bruteforce/.

[36] Lejla Batina, Patrick Jauernig, Nele Mentens, Ahmad-Reza Sadeghi, and Emmanuel Stapf. "In Hardware We Trust: Gains and Pains of Hardware-assisted Security". In: *Design Automation Conference, DAC 2019*. ACM, p. 44. DOI: 10.1145/3316781.3323480.

[37] Christopher Bellman and Paul C. van Oorschot. "Analysis, Implications, and Challenges of an Evolving Consumer IoT Security Landscape". In: *International Conference on Privacy, Security and Trust, PST 2019*. IEEE, pp. 1–7. DOI: 10.1109/PST47121.2019.8949058.

[38] Daniel J. Bernstein, Bernard van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. "TweetNaCl: A Crypto Library in 100 Tweets". In: *LATINCRYPT*. Springer LNCS 8895. 2014, pp. 64–83. DOI: 10.1007/978-3-319-16295-9_4.

[39] Cliff Biffle. *lilos*. URL: https://github.com/cbiffle/lilos.

[40] Simon Biggs, Damon Lee, and Gernot Heiser. "The Jury Is In: Monolithic OS Design Is Flawed: Microkernel-based Designs Improve Security". In: *Asia-Pacific Workshop on Systems, APSys 2018*, 16:1–16:7. DOI: 10.1145/3265723.3265733.

[41] *Bluetooth Core Specification*. Version 5.2. Bluetooth Special Interest Group. Dec. 2019.

[42] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. "Theseus: an Experiment in Operating System Structure and State Management". In: *OSDI*. USENIX Association, Nov. 2020, pp. 1–19.

[43] Kevin Boos, Emilio Del Vecchio, and Lin Zhong. "A Characterization of State Spill in Modern Operating Systems". In: *EuroSys*. ACM, 2017, pp. 389–404. DOI: 10.1145/3064176.3064205.

[44] C. Bormann, M. Ersue, and A. Keranen. *Terminology for Constrained-Node Networks*. RFC 7228. May 2014. URL: http://www.rfc-editor.org/rfc/rfc7228.txt.

[45] C. Bormann and P. Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 7049. Oct. 2013. URL: http://www.rfc-editor.org/rfc/rfc7049.txt.

[46] A. Brandt and J. Buron. *Transmission of IPv6 Packets over ITU-T G.9959 Networks*. RFC 7428. Feb. 2015. URL: http://www.rfc-editor.org/rfc/rfc7428.txt.

[47] Buildroot Contributors. *Buildroot*. URL: https://buildroot.org/.

[48] Guofa Cai, Yi Fang, Jinming Wen, Guojun Han, and Xiaodong Yang. "QoS-Aware Buffer-Aided Relaying Implant WBAN for Healthcare IoT: Opportunities and Challenges". In: *IEEE Netw.* 33.4 (2019), pp. 96–103. DOI: 10.1109/MNET.2019.1800405.

[49] Holly Chiang, Hudson Ayers, Daniel B. Giffin, Amit Levy, and Philip Levis. "Power Clocks: Dynamic Multi-Clock Management for Embedded Systems". In: *Embedded Wireless Systems and Networks, EWSN*. ACM, 2021, pp. 139–150.

[50] Will Crichton. *The Usability of Ownership*. 2020. arXiv: 2011.06171.

[51] John Criswell, Nicolas Geoffray, and Vikram S. Adve. "Memory Safety for Low-Level Software/Hardware Interactions". In: *USENIX Security Symposium*. 2009, pp. 83–100.

[52] Cody Cutler, M. Frans Kaashoek, and Robert Tappan Morris. "The benefits and costs of writing a POSIX kernel in a high-level language". In: *OSDI*. USENIX Association, 2018, pp. 89–105.

[53] Stephen E. Deering and Robert M. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460. Dec. 1998. URL: http://www.rfc-editor.org/rfc/rfc2460.txt.

[54] Peter J. Denning. "A Short Theory of Multiprogramming". In: *Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems, MASCOTS*. IEEE Computer Society, 1995, pp. 2–7. DOI: 10.1109/MASCOT.1995.378718.

[55] Dinakar Dhurjati, Sumant Kowshik, Vikram S. Adve, and Chris Lattner. "Memory safety without garbage collection for embedded applications". In: *ACM Trans. Embed. Comput. Syst.* 4.1 (2005), pp. 73–111. DOI: 10.1145/1053271.1053275.

[56] F. John Dian, Reza Vahidnia, and Alireza Rahmati. "Wearables and the Internet of Things (IoT), Applications, Opportunities, and Challenges: A Survey". In: *IEEE Access* 8 (2020), pp. 69200–69211. DOI: 10.1109/ACCESS.2020.2986329.

[57] M. Dohler, T. Watteyne, T. Winter, and D. Barthel. *Routing Requirements for Urban Low-Power and Lossy Networks*. RFC 5548. May 2009. URL: https://www.ietf.org/rfc/rfc5548.txt.

[58] Danny Dolev and Andrew Chi-Chih Yao. "On the security of public key protocols". In: *IEEE Trans. Inf. Theory* 29.2 (1983), pp. 198–207. DOI: 10.1109/TIT.1983.1056650.

[59] Electronics360. *Fitbit Ionic Teardown*. URL: https://electronics360.globalspec.com/article/11796/teardown-fitbit-ionic-fb503.

[60] *Fitbit Flex User Manual*. Version 1.1. Fitbit.

[61] *Fitbit Ionic User Manual*. Version 4.5. Fitbit.

[62] Aurélien Francillon and Claude Castelluccia. "Code injection attacks on harvard-architecture devices". In: *CCS*. ACM, 2008, pp. 15–26. DOI: 10.1145/1455770.1455775.

[63] Walter Galan. *Fitbit Flex Teardown*. URL: https://www.ifixit.com/Teardown/Fitbit+Flex+Teardown/16050.

[64] Catherine H. Gebotys. "Side Channel Attacks on the Embedded System". In: *Security in Embedded Devices*. Springer US, 2010, pp. 163–222. DOI: 10.1007/978-1-4419-1530-6_8.

[65] Travis Goodspeed. *Stack Overflow Exploits for Wireless Sensor Networks Over 802.15.4*. Feb. 2008. URL: http://osgug.ucaiug.org/utilisec/amisec/Meetings/20080822%20-%20Face-to-Face%20Consumers%20Energy%20%20(%20Jackson,%20MI)/Travis_Goodspeed_tidc08.pdf.

[66] Travis Goodspeed. *Cold, Lableless HNO3 Decapping Procedure*. URL: http://travisgoodspeed.blogspot.com/2009/06/cold-lableless-hno3-decapping-procedure.html.

[67] Google. *OpenThread*. URL: https://github.com/openthread/openthread.

[68] Peter Leo Gorski, Luigi Lo Iacono, Dominik Wermke, Christian Stransky, Sebastian Möller, Yasemin Acar, and Sascha Fahl. "Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic API Misuse". In: *Symposium on Usable Privacy and Security, SOUPS*. USENIX Association, 2018, pp. 265–281.

[69] *Trusted Platform Module Library Part 1: Architecture*. Version Level 00 Revision 01.59. Trusted Computing Group (TCG). Nov. 2019.

[70] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes. "Operating Systems for Low-End Devices in the Internet of Things: A Survey". In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 720–734. DOI: 10.1109/JIOT.2015.2505901.

[71] Tyler Hall. *Rust on Zephyr RTOS*. URL: https://github.com/tylerwhall/zephyr-rust.

[72] Per Brinch Hansen. "The nucleus of a multiprogramming system". In: *Commun. ACM* 13.4 (1970), pp. 238–241. DOI: 10.1145/362258.362278.

[73] Taylor Hardin, Ryan Scott, Patrick Proctor, Josiah D. Hester, Jacob Sorber, and David Kotz. "Application Memory Isolation on Ultra-Low-Power MCUs". In: *USENIX*. USENIX Association, 2018, pp. 127–132.

[74] Gernot Heiser. "Secure Embedded Systems Need Microkernels". In: *login Usenix Mag.* 30.6 (2005).

[75] Hideto Hidaka. *Embedded Flash Memory for Embedded Systems: Technology, Design for Sub-systems, and Innovations*. Springer, 2018.

[76] Jaap-Henk Hoepman and Bart Jacobs. "Increased Security through Open Source". In: *Commun. ACM* 50.1 (Jan. 2007), pp. 79–83. ISSN: 0001-0782. DOI: 10.1145/1188913.1188921.

[77] Yong-Geun Hong, Carles Gomez, Abdur Rashid Sangi, and Samita Chakrabarti. *IPv6 over Constrained Node Networks (6lo) Applicability & Use cases*. Internet-Draft draft-ietf-6lo-use-cases-10. Work in Progress. IETF. URL: https://datatracker.ietf.org/doc/html/draft-ietf-6lo-use-cases-10.

[78] Mei-Chen Hsueh, T. K. Tsai, and R. K. Iyer. "Fault injection techniques and tools". In: *Computer* 30.4 (1997), pp. 75–82. DOI: 10.1109/2.585157.

[79] Jonathan W. Hui and David E. Culler. "IP is dead, long live IP for wireless sensor networks". In: *SenSys*. ACM, 2008, pp. 15–28. DOI: 10.1145/1460412.1460415.

[80] Galen C. Hunt and James R. Larus. "Singularity: rethinking the software stack". In: *ACM SIGOPS Oper. Syst. Rev.* 41.2 (2007), pp. 37–49. DOI: 10.1145/1243418.1243424.

[81] "IEEE Standard for Low-Rate Wireless Networks". In: *IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015)* (2020), pp. 1–800. DOI: 10.1109/IEEESTD.2020.9144691.

[82] "IEEE Standard for Test Access Port and Boundary-Scan Architecture". In: *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)* (2013), pp. 1–444. DOI: 10.1109/IEEESTD.2013.6515989.

[83] *Intel 64 and IA-32 Architectures Software Developer's Manual:* Volume 3: System Programming Guide. Intel. June 2021.

[84] Intel Corporation. *PCI Express Architecture Resources*. URL: https://www.intel.com/content/www/us/en/io/pci-express/pci-express-architecture-devnet-resources.html.

[85] M. Aiman Ismail and Thomas C. Schmidt. "A DTLS Abstraction Layer for the Recursive Networking Architecture in RIOT". In: *CoRR* abs/1906.12143 (2019). arXiv: 1906.12143.

[86] Bruce L. Jacob and Trevor N. Mudge. "Virtual Memory: Issues of Implementation". In: *Computer* 31.6 (1998), pp. 33–43. DOI: 10.1109/2.683005.

[87] Trent Jaeger. *Operating System Security*. Morgan & Claypool Publishers, 2008.

[88] Farhana Javed, Muhammad Khalil Afzal, Muhammad Sharif, and Byung-Seo Kim. "Internet of Things (IoT) Operating Systems Support, Networking Technologies, Applications, and Challenges: A Comparative Review". In: *IEEE Communications Surveys & Tutorials* 20.3 (2018), pp. 2062–2100. DOI: 10.1109/COMST.2018.2817685.

[89] S. J. Johnston, M. Apetroaie-Cristea, M. Scott, and S. J. Cox. "Applicability of commodity, low cost, single board computers for Internet of Things devices". In: *IEEE World Forum on Internet of Things (WF-IoT)*. 2016, pp. 141–146. DOI: 10.1109/WF-IoT.2016.7845414.

[90] Marc Joye. "Basics of Side-Channel Analysis". In: *Cryptographic Engineering*. Springer, 2009, pp. 365–380. DOI: 10.1007/978-0-387-71817-0_13.

[91] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: securing the foundations of the rust programming language". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 66:1–66:34. DOI: 10.1145/3158154.

[92] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "Safe Systems Programming in Rust: The Promise and the Challenge". In: (2020). URL: https://robbertkrebbers.nl/research/articles/safe_programming_rust.pdf.

[93] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "Safe Systems Programming in Rust". In: *Commun. ACM* 64.4 (Mar. 2021), pp. 144–152. ISSN: 0001-0782. DOI: 10.1145/3418295.

[94] Fatma Karray, Mohamed Wassim Jmal, Alberto García Ortiz, Mohamed Abid, and Abdulfattah Mohammad Obeid. "A comprehensive survey on wireless sensor node hardware platforms". In: *Comput. Networks* 144 (2018), pp. 89–110. DOI: 10.1016/j.comnet.2018.05.010.

[95] Hayder A. A. Al-Kashoash, Harith Kharrufa, Yaarob Al- Nidawi, and Andrew H. Kemp. "Congestion control in wireless sensor and 6LoWPAN networks: toward the Internet of Things". In: *Wirel. Networks* 25.8 (2019), pp. 4493–4522. DOI: 10.1007/s11276-018-1743-y.

[96] Shapla Khanam, Ismail Bin Ahmedy, Mohd Yamani Idna Bin Idris, Mohamed Hisham Jaward, and Aznul Qalid Md Sabri. "A Survey of Security Challenges, Attacks Taxonomy and Advanced Countermeasures in the Internet of Things". In: *IEEE Access* 8 (2020), pp. 219709–219743. DOI: 10.1109/ACCESS.2020.3037359.

[97] Peter Kietzmann, Lena Boeckmann, Leandro Lanzieri, Thomas C. Schmidt, and Matthias Wä hlisch. "A Performance Study of Crypto-Hardware in the Low-end IoT". In: *IACR Cryptol. ePrint Arch.* (2021), p. 58. URL: https://eprint.iacr.org/2021/058.

[98] Dongkwan Kim, Suwan Park, Kibum Choi, and Yongdae Kim. "BurnFit: Analyzing and Exploiting Wearable Devices". In: *Information Security Applications - International Workshop, WISA*. Vol. 9503. Lecture Notes in Computer Science. Springer, 2015, pp. 227–239. DOI: 10.1007/978-3-319-31875-2_19.

[99] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2019. URL: https://doc.rust-lang.org/stable/book/.

[100] Ramesh Krishnamoorthy, Kalimuthu Krishnan, Bharatiraja Chokkalingam, Sanjeevikumar Padmanaban, Zbigniew Leonowicz, Jens Bo Holm-Nielsen, and Massimo Mitolo. "Systematic Approach for State-of-the-Art Architectures and System-on-Chip Selection for Heterogeneous IoT Applications". In: *IEEE Access* 9 (2021), pp. 25594–25622. DOI: 10.1109/ACCESS.2021.3055650.

[101] N. Kushalnagar, G. Montenegro, and C. Schumacher. *IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Prob-*

lem Statement, and Goals. RFC 4919. Aug. 2007. URL: http://www.rfc-editor.org/rfc/rfc4919.txt.

[102] A. Langley, W. Chang, N. Mavrogiannopoulos, J. Strombergson, and S. Josefsson. *ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)*. RFC 7905. June 2016. URL: https://www.rfc-editor.org/rfc/rfc7905.txt.

[103] Martine Lenders. "Analysis and Comparison of Embedded Network Stacks: Design and Evaluation of the GNRC Network Stack". MA thesis. Freie Universität Berlin, 2016. URL: http://doc.riot-os.org/mlenders_msc.pdf.

[104] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. "Multiprogramming a 64kB Computer Safely and Efficiently". In: *Symposium on Operating Systems Principles, SOSP*. ACM, Oct. 2017, pp. 234–251. DOI: 10.1145/3132747.3132786.

[105] Amit Aryeh Levy. "A Secure Operating System for the Internet of Things". PhD thesis. Stanford University, 2018.

[106] Jialin Li, Samantha Miller, Danyang Zhuo, Ang Chen, Jon Howell, and Thomas Anderson. "An Incremental Path Towards a Safer OS Kernel". In: HotOS. May 2021.

[107] Lobaro. *FreeRTOS-Rust*. URL: https://github.com/lobaro/FreeRTOS-rust.

[108] lowRISC Contributors. *OpenTitan Firmware Update*. URL: https://docs.opentitan.org/doc/security/specs/firmware_update/.

[109] lowRISC Contributors. *OpenTitan Logical Security Model*. URL: https://docs.opentitan.org/doc/security/logical_security_model/.

[110] lowRISC Contributors. *OpenTitan Secure Boot*. URL: https://docs.opentitan.org/doc/security/specs/secure_boot/.

[111] Lan Luo, Yue Zhang, Cliff C. Zou, Xinhui Shao, Zhen Ling, and Xinwen Fu. "On Runtime Software Security of TrustZone-M based IoT Devices". In: *CoRR* abs/2007.05876 (2020). arXiv: 2007.05876.

[112] Ken MacKay. *micro-ecc*. https://github.com/kmackay/micro-ecc.

[113] Pieter Maene. "Lightweight Roots of Trust for Modern Systems-on-Chip". PhD thesis. KU Leuven, Oct. 2019.

[114] Pieter Maene, Johannes Götzfried, Ruan de Clercq, Tilo Müller, Felix C. Freiling, and Ingrid Verbauwhede. "Hardware-Based Trusted Computing Architectures for Isolation and Attestation". In: *IEEE Trans. Computers* 67.3 (2018), pp. 361–374. DOI: 10.1109/TC.2017.2647955.

[115] Pieter Maene, Johannes Götzfried, Tilo Müller, Ruan de Clercq, Felix C. Freiling, and Ingrid Verbauwhede. "Atlas: Application Confidentiality in Compromised Embedded Systems". In: *IEEE Trans. Dependable Secur. Comput.* 16.3 (2019), pp. 415–423. DOI: 10.1109/TDSC.2018.2858257.

[116] Francine Mäkelä and Johan Lindskogen. "Defining a minimal BLE stack: A Bluetooth Low Energy implementation in Rust". MA thesis. University of Gothenburg, 2018.

[117] Nicholas D. Matsakis and Felix S. Klock II. "The Rust language". In: *ACM conference on High integrity language technology, HILT 2014*, pp. 103–104. DOI: 10.1145/2663171.2663188.

[118] Sebastian Meiling and Thomas C. Schmidt. "LoRa in the Field: Insights from Networking the Smart City Hamburg with RIOT". In: *CoRR* abs/2003.08647 (2020). arXiv: 2003.08647.

[119] *AVR Instruction Set Manual*. Version DS40002198B. Microchip Technology. Feb. 2021.

[120] Daniele Midi, Mathias Payer, and Elisa Bertino. "Memory Safety for Embedded Devices with nesCheck". In: *Asia Conference on Computer and Communications Security, AsiaCCS*. ACM, 2017, pp. 127–139. DOI: 10.1145/3052973.3053014.

[121] Matt Miller. *Pursuing Durable Safety for Systems Software*. SSTIC2020. URL: https://www.sstic.org/2020/presentation/ouverture_2020/.

[122] R. Min, M. Bhardwaj, Seong-Hwan Cho, N. Ickes, E. Shih, A. Sinha, Alice Wang, and A. Chandrakasan. "Energy-centric enabling tecumologies for wireless sensor networks". In: *IEEE Wireless Communications* 9.4 (2002), pp. 28–39. DOI: 10.1109/MWC.2002.1028875.

[123] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. RFC 4944. Sept. 2007. URL: http://www.rfc-editor.org/rfc/rfc4944.txt.

[124] B. Moran, H. Tschofenig, D. Brown, and M. Meriac. *A Firmware Update Architecture for Internet of Things*. RFC 9019. Apr. 2021. URL: http://www.rfc-editor.org/rfc/rfc9019.txt.

[125] Brendan Moran, Hannes Tschofenig, Henk Birkholz, and Koen Zandberg. *A Concise Binary Object Representation (CBOR)-based Serialization Format for the Software Updates for Internet of Things (SUIT) Manifest*. Internet-Draft draft-ietf-suit-manifest-09. Work in Progress. IETF. URL: https://datatracker.ietf.org/doc/html/draft-ietf-suit-manifest-09.

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

[127] Mike Muller. *IoT Security: The Ugly Truth*. The Internet of Things Security Summit 2015, Bletchley Park. URL: https://www.iotsecurityfoundation.org/iot-security-summit-2015/.

[128] Arslan Musaddiq, Yousaf Bin Zikria, Oliver Hahm, Heejung Yu, Ali Kashif Bashir, and Sung Won Kim. "A Survey on Resource Management in IoT Operating Systems". In: *IEEE Access* 6 (2018), pp. 8459–8482. DOI: 10.1109/ACCESS.2018.2808324.

[129] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. "RedLeaf: Isolation and Communication in a Safe Operating System". In: *OSDI*. USENIX Association, Nov. 2020, pp. 21–39.

[130] Shoei Nashimoto, Daisuke Suzuki, Rei Ueno, and Naofumi Homma. "Bypassing Isolated Execution on RISC-V with Fault Injection". In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 1193. URL: https://eprint.iacr.org/2020/1193.

[131] J. Nieminen, T. Savolainen, M. Isomaki, B. Patil, Z. Shelby, and C. Gomez. *IPv6 over BLUETOOTH(R) Low Energy*. RFC 7668. Oct. 2015. URL: http://www.rfc-editor.org/rfc/rfc7668.txt.

[132] Filip Nilsson and Sebastian Lund. "Abstraction Layers and Energy Efficiencyin TockOS, a Rust-based Runtime for the Internet of Things". MA thesis. University of Gothenburg, 2018.

[133] New Jersey Cybersecurity & Communications Integration Cell (NJCCIC). *Hajime Botnet*. URL: https://www.cyber.nj.gov/threat-center/threat-profiles/botnet-variants/hajime-botnet.

[134] *nRF24L01+ Preliminary Product Specification*. Version 1.0. Nordic Semiconductor. Mar. 2008.

[135] *nRF52832 Product Specification*. Version 1.4. Nordic Semiconductor. Oct. 2017.

[136] Roman Obermaisser, Philipp Peti, and Fulvio Tagliabo. "An integrated architecture for future car generations". In: *Real Time Syst.* 36.1-2 (2007), pp. 101–133. DOI: 10.1007/s11241-007-9015-4.

[137] Miguel Ojeda. "Rust for Linux". In: RFC. 2021. URL: https://lkml.org/lkml/2021/4/14/1023.

[138] Mike Oluwatayo Ojo, Stefano Giordano, Gregorio Procissi, and Ilias Nektarios Seitanidis. "A Review of Low-End, Middle-End, and High-End Iot Devices". In: *IEEE Access* 6 (2018), pp. 70528–70554. DOI: 10.1109/ACCESS.2018.2879615.

[139] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, and Christof Fetzer. "Fex: A Software Systems Evaluator". In: *Dependable Systems and Networks, DSN*. IEEE Computer Society, 2017, pp. 543–550. DOI: 10.1109/DSN.2017.25.

[140] Paul C. van Oorschot. *Computer Security and the Internet: Tools and Jewels*. Springer, 2020.

[141] OSDev Wiki. *Stack Smashing Protector*. URL: https://wiki.osdev.org/Stack_Smashing_Protector.

[142] Pat Pannuto, Michael P. Andersen, Tom Bauer, Bradford Campbell, Amit Levy, David E. Culler, Philip Levis, and Prabal Dutta. "A networked embedded system platform for the post-mote era". In: *Conference on Embedded Network Sensor Systems, SenSys*. ACM, 2014, pp. 354–355. DOI: 10.1145/2668332.2668364.

[143] D. Papp, Z. Ma, and L. Buttyan. "Embedded systems security: Threats, vulnerabilities, and attack taxonomy". In: *Annual Conference on Privacy, Security and Trust (PST)*. 2015, pp. 145–152. DOI: 10.1109/PST.2015.7232966.

[144] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. *Bootstrapping Trust in Modern Computers*. Vol. 10. Springer Briefs in Computer Science. Springer, 2011. DOI: 10.1007/978-1-4614-1460-5.

[145] Sean Peisert, Bruce Schneier, Hamed Okhravi, Fabio Massacci, Terry Benzel, Carl E. Landwehr, Mohammad Mannan, Jelena Mirkovic, Atul Prakash, and James Bret Michael. "Perspectives on the SolarWinds Incident". In: *IEEE Secur. Priv.* 19.2 (2021), pp. 7–13. DOI: 10.1109/MSEC.2021.3051235.

[146] J. Postel. *User Datagram Protocol*. STD 6. Aug. 1980. URL: http://www.rfc-editor.org/rfc/rfc768.txt.

[147] Paolo Prinetto and Gianluca Roascio. "Hardware Security, Vulnerabilities, and Attacks: A Comprehensive Taxonomy". In: *Italian Conference on Cyber Security*. Vol. 2597. CEUR Workshop Proceedings. 2020, pp. 177–189. URL: http://ceur-ws.org/Vol-2597/paper-16.pdf.

[148] Weizhong Qiang, Kang Zhang, and Hai Jin. "Reducing TCB of Linux Kernel Using User-Space Device Driver". In: *Algorithms and Architectures for Parallel Processing, ICA3PP*. Vol. 10048. Springer, 2016, pp. 572–585. DOI: 10.1007/978-3-319-49583-5_45.

[149] Panagiotis I. Radoglou-Grammatikis, Panagiotis G. Sarigiannidis, and Ioannis D. Moscholios. "Securing the Internet of Things: Challenges, threats and solutions". In: *Internet of Things* 5 (2019), pp. 41–70. DOI: 10.1016/j.iot.2018.11.003.

[150] Redox Developers. *Redox OS*. URL: https://www.redox-os.org/.

[151] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. URL: http://www.rfc-editor.org/rfc/rfc8446.txt.

[152] E. Rescorla and N. Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347. Jan. 2012. URL: http://www.rfc-editor.org/rfc/rfc6347.txt.

[153] Kudelski Security Research. *SWD – ARM's alternative to JTAG*. URL: https://research.kudelskisecurity.com/2019/05/16/swd-arms-alternative-to-jtag/.

[154] RIOT Contributors. *Networking libraries*. URL: https://doc.riot-os.org/group__net.html.

[155] RIOT Contributors. *RIOT Contributors: Coding Conventions*. URL: https://github.com/RIOT-OS/RIOT/blob/master/CODING_CONVENTIONS.md. (accessed: 19.11.2020).

[156] RIOT Contributors. *RIOT Contributors: RIOT in a nutshell*. URL: https://doc.riot-os.org/.

[157] RIOT Contributors. *RIOT Design Goals*. URL: https://github.com/RIOT-OS/RIOT/blob/master/doc/memos/rdm0001.md.

[158] RIOT Contributors. *RIOT Source Code Repository*. URL: https://github.com/RIOT-OS/RIOT.

[159] RIOT. *RIOT Summit*. URL: https://summit.riot-os.org/.

[160] RIOT Contributors. *riotboot*. Source Code. URL: https://github.com/RIOT-OS/RIOT/tree/master/sys/riotboot.

[161] Christian Amsüss. *Rust wrappers around the RIOT operating system*. URL: https://gitlab.com/etonomy/riot-wrappers.

[162] RIOT Contributors. *SUIT secure firmware OTA upgrade infrastructure*. URL: https://doc.riot-os.org/group__sys__suit.html.

[163] Andrew Waterman, Yunsup Lee, David Patterson, and Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*. Version 1.0. Electrical Engineering and Computer Sciences University of California at Berkeley. May 2011.

[164] Andrew Waterman and Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. Version 20191213. RISC-V Foundation. Dec. 2019.

[165] Andrew Waterman and Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. Version 20190608-Priv-MSU-Ratified. RISC-V Foundation. June 2019.

[166] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. "Return-Oriented Programming: Systems, Languages, and Applications". In: *ACM Trans. Inf. Syst. Secur.* 15.1 (2012), 2:1–2:34. DOI: 10.1145/2133375.2133377.

[167] Eyal Ronen, Colin O'Flynn, Adi Shamir, and Achi-Or Weingarten. "IoT Goes Nuclear: Creating a ZigBee Chain Reaction". In: *IACR Cryptol. ePrint Arch.* (2016), p. 1047. URL: http://eprint.iacr.org/2016/1047.

[168] RTIC Contributors. *Real-Time Interrupt-driven Concurrency (RTIC)*. URL: https://rtic.rs/0.5/book/en/.

[169] Rust Crypto Github Organization. *Rust Crypto*. URL: https://github.com/RustCrypto.

[170] Rust Developers. *The Rust Core Library*. URL: https://doc.rust-lang.org/core/.

[171] Rust Developers. *The Rust Standard Library*. URL: https://doc.rust-lang.org/std/.

[172] J. Schaad. *CBOR Object Signing and Encryption (COSE)*. RFC 8152. July 2017. URL: http://www.rfc-editor.org/rfc/rfc8152.txt.

[173] seL4 Project. *The seL4 Microkernel*. URL: https://sel4.systems/.

[174] Nordic Semiconductor. *nRF52 DK*. URL: https://www.nordicsemi.com/Software-and-Tools/Development-Kits/nRF52-DK.

[175] *SX1276-7-8-9 Datasheet*. Version 7. Semtech. May 2020.

[176] Abderrahmane Sensaoui, Oum-El-Kheir Aktouf, David H ély, and Stephane Di Vito. "An In-depth Study of MPU-Based Isolation Techniques". In: *J. Hardw. Syst. Secur.* 3.4 (2019), pp. 365–381. DOI: 10.1007/s41635-019-00078-6.

[177] Z. Shelby, K. Hartke, and C. Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. June 2014. URL: http://www.rfc-editor.org/rfc/rfc7252.txt.

[178] SiFive. *HiFive1 Rev B*. URL: https://www.sifive.com/boards/hifive1-rev-b.

[179] Miguel Silva, David Cerdeira, Sandro Pinto, and Tiago Gomes. "Operating Systems for Internet of Things Low-End Devices: Analysis and Benchmarking". In: *IEEE Internet Things J.* 6.6 (2019), pp. 10375–10383. DOI: 10.1109/JIOT.2019.2939008.

[180] Larry Singleton, Rui Zhao, Myoungkyu Song, and Harvey P. Siy. "CryptoTutor: Teaching Secure Coding Practices through Misuse Pattern Detection". In: *Conference on Information Technology Education, SIGITE*. ACM, 2020, pp. 403–408. DOI: 10.1145/3368308.3415419.

[181] Yang Song. "On Control Flow Hijacks of unsafe Rust". 2017.

[182] Statista. *Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025*. URL: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/.

[183] Marc Stiegler. *The E Language in a Walnut: Capabilities*. URL: http://www.skyhunter.com/marcs/ewalnut.html#SEC42.

[184] stm32-rs Contributors. *STM32 Peripheral Access Crates*. URL: https://github.com/stm32-rs/stm32-rs.

[185] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. "Recovering device drivers". In: *ACM Trans. Comput. Syst.* 24.4 (2006), pp. 333–360. DOI: 10.1145/1189256.1189257.

[186] Laszlo Szekeres, Mathias Payer, Tao Wei, and R. Sekar. "Eternal War in Memory". In: *IEEE Secur. Priv.* 12.3 (2014), pp. 45–53. DOI: 10.1109/MSP.2014.44.

[187] Andrew Tanenbaum and Herbert Bos. *Modern Operating Systems*. Pearson, Mar. 2014.

[188] Tock Contributors. *Tock C Userland Source Code Repository*. URL: https://github.com/tock/libtock-c.

[189] Tock Contributors. *Tock Documentation: Memory Isolation*. URL: https://github.com/tock/tock/blob/master/doc/Memory_Isolation.md.

[190] Tock Contributors. *Tock Documentation: Memory Layout*. URL: https://github.com/tock/tock/blob/master/doc/Memory_Layout.md.

[191] Tock Contributors. *Tock Documentation: Networking Stack Design Document*. URL: https://github.com/tock/tock/blob/master/doc/Networking_Stack.md.

[192] Tock Contributors. *Tock Documentation: Syscalls*. URL: https://github.com/tock/tock/blob/master/doc/Syscalls.md.

[193] Tock Contributors. *Tock Documentation: Tock Binary Format*. URL: https://github.com/tock/tock/blob/master/doc/TockBinaryFormat.md.

[194] Tock Contributors. *Tock Documentation: Tock Design*. URL: https://github.com/tock/tock/blob/master/doc/Design.md.

[195] Tock Contributors. *Tock Documentation: Tock Overview*. URL: https://github.com/tock/tock/blob/master/doc/Overview.md.

[196] Tock Contributors. *Tock Reference Document (TRD): System Calls*. URL: https://github.com/tock/tock/blob/master/doc/reference/trd104-syscalls.md.

[197] Tock Contributors. *Tock Rust Userland Source Code Repository*. URL: https://github.com/tock/libtock-rs.

[198] Tock Contributors. *Tock Source Code Repository*. URL: https://github.com/tock/tock.

[199] *TZ1200 Series Hardware Specification*. Version 1.4. Toshiba. Jan. 2018.

[200] Wade Trappe, Richard E. Howard, and Robert S. Moore. "Low-Energy Security: Limits and Opportunities in the Internet of Things". In: *IEEE Secur. Priv.* 13.1 (2015), pp. 14–21. DOI: 10.1109/MSP.2015.7.

[201] Hannes Tschofenig and Emmanuel Baccelli. "Cyberphysical Security for the Masses: A Survey of the Internet Protocol Suite for Internet of Things Security". In: *IEEE Secur. Priv.* 17.5 (2019), pp. 47–57. DOI: 10.1109/MSEC.2019.2923973.

[202] Valentine Valyaeff. *Drone OS*. URL: https://www.drone-os.com/.

[203] Jiliang Wang, Feng Hu, Ye Zhou, Yunhao Liu, Hanyi Zhang, and Zhe Liu. "BlueDoor: breaking the secure information flow via BLE vulnerability". In: *MobiSys*. ACM, 2020, pp. 286–298. DOI: 10.1145/3386901.3389025.

[204] Hans Weigand and Philip I. Elsas. "Auditability as a Design Problem". In: *Conference on Business Informatics (CBI)*. IEEE, 2019, pp. 276–285. DOI: 10.1109/CBI.2019.00038.

[205] Hans Weigand, Paul Johannesson, Birger Andersson, Maria Bergholtz, and Faiza Allah Bukhsh. "Conceptualizing Auditability". In: *Conference on Advanced Information Systems Engineering, CAiSE*. Vol. 998. CEUR Workshop Proceedings. 2013, pp. 49–56. URL: http://ceur-ws.org/Vol-998/Paper07.pdf.

[206] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad- Reza Sadeghi. "TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V". In: *NDSS*. The Internet Society, 2019.

[207] Johnathan Van Why. *Tock Documentation: Tock Threat Model*. URL: https://github.com/tock/tock/tree/master/doc/threat_model.

[208] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. "RIPE: runtime intrusion prevention evaluator". In: *Annual Computer Security Applications Conference, ACSAC*. ACM, 2011, pp. 41–50. DOI: 10.1145/2076732.2076739.

[209] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, JP. Vasseur, and R. Alexander. *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks*. RFC 6550. Mar. 2012. URL: http://www.rfc-editor.org/rfc/rfc6550.txt.

[210] Marilyn Wolf and Dimitrios N. Serpanos. "Safety and Security in Cyber-Physical Systems and Internet-of-Things Systems". In: *Proc. IEEE* 106.1 (2018), pp. 9–20. DOI: 10.1109/JPROC.2017.2781198.

[211] Hongyan Xia et al. "CheriRTOS: A Capability Model for Embedded Devices". In: *International Conference on Computer Design, ICCD*. IEEE Computer Society, 2018, pp. 92–99. DOI: 10.1109/ICCD.2018.00023.

[212] Carter Yagemann, Matthew Pruett, Simon P. Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. "ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems". In: *USENIX*. Aug. 2021.

[213] Fan Yang. "Towards Deeply Reconfigurable, Long-Life Internet of Things Platforms". PhD thesis. KU Leuven, Jan. 2019.

[214] Zengxu Yang and C. Hwa Chang. "6LoWPAN Overview and Implementations". In: *Embedded Wireless Systems and Networks, EWSN*. ACM, 2019, pp. 357–361. URL: https://dl.acm.org/citation.cfm?id=3324409.

[215] Yocto Project Contributors. *Yocto Project*. URL: https://www.yoctoproject.org/.

[216] YVT. *R3*. URL: https://github.com/yvt/r3.

[217] Koen Zandberg, Kaspar Schleiser, Francisco Acosta Padilla, Hannes Tschofenig, and Emmanuel Baccelli. "Secure Firmware Updates for Constrained IoT Devices Using Open Standards: A Reality Check". In: *IEEE Access* 7 (2019), pp. 71907–71920. DOI: 10.1109/ACCESS.2019.2919760.

[218] Zephyr Project. *Zephyr OS*. URL: https://www.zephyrproject.org/.

[219] Liang Zhao and Guangwen Wang. "Research Status of 6LoWPAN in the Field of Internet of Things". In: *Conference on Automation, Control and Robotics*

Engineering, CACRE. IEEE, 2020, pp. 739–743. DOI: 10.1109/CACRE50138.2020.9230293.

[220] Sun Zhou and Jun Chen. "Experimental Evaluation of the Defense Capability of ARM-based Systems against Buffer Overflow Attacks in Wireless Networks". In: *International Conference on Electronics Information and Emergency Communication (ICEIEC)*. IEEE, 2020, pp. 375–378. DOI: 10.1109/ICEIEC49280.2020.9152302.

[221] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. "Good Motive but Bad Design: Why ARM MPU Has Become an Outcast in Embedded Systems". In: *CoRR* abs/1908.03638 (2019). arXiv: 1908.03638. URL: http://arxiv.org/abs/1908.03638.

[222] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. "HACL*: A Verified Modern Cryptographic Library". In: *ACM, CCS 2017*, pp. 1789–1806. DOI: 10.1145/3133956.3134043.

Appendices

Appendix A.

Technical Details for E5 on RIOT

This section contains the particular details of experiment E5 conducted on RIOT. This experiment involves exploitation of a function containing a stack-based overflow vulnerability with PMP enabled in RIOT, making the stack non-executable. In turn, this should prevent code execution on the stack. The goal is to determine whether we may circumvent PMP by disabling it entirely via a code reuse attack, followed right after with additional arbitrary code execution on the–now unprotected–stack.

```
1  @ address          assembly code
2  @ ----------       ------------
3    0x000000dc       push {r7, lr}
4    0x000000de       sub sp, #56
5    0x000000e0       add r7, sp, #0
6    @ ...
7    0x000000ea       bl 0x1f32 <memcpy>
8    @ ...
9    0x000000f0       adds r7, #56
10   0x000000f2       mov sp, r7
11   0x000000f4       pop {r7, pc}
```

Listing A.1: Disassembly snippet of `vulnerable()`

The disassembly of the `vulnerable()` function in Listing A is where we may start. The function prologue comprised by lines 3-5 sets up the stack frame and stores the current stack pointer into register R7, this will be important later. On line 7, another function (`memcpy()`) is invoked, the details of this function are not relevant, however, recalling the fact that the `bl` instruction on line 7 will update the link register (LR) with an appropriate return address will be useful later. In this case, the return address will be $0x000000ee$, i.e., the subsequent instruction of `vulnerable()`. Following, lines 9-11 comprise `vulnerable()`'s epilogue, where the last line (line 11) involves loading values from the stack into the R7 and PC registers. Just as before, we take advantage of this fact to overwrite the address to be stored into PC via a stack-based buffer overflow to alter execution flow. To achieve thus, we inject the modified attack string given in Listing A.2.

```
\x31\x08\x00\x20\x09\x46\x09\x46\x09\x46\x09\x46\x09\x46\x4f\xf0
\x05\x04\x4f\xea\x04\x71\x4f\xea\x04\x22\x02\xf1\x04\x02\x11\x44
\x51\xf8\x04\x2b\x4f\xf0\x01\x03\x4f\xea\x43\x43\x82\xea\x03\x02
\x41\xf8\x04\x2c\xf1\x07\x00\x20\xd5\x0b\x00\x00
```

Listing A.2: Modified attack string containing the address of one of the NOP instructions on the stack (yellow), 5 NOPs (green), 10 assembly instructions (blue), precisely chosen R7 value (orange), followed by the starting address of `mpu_disable()` (red)

This time around, the PC register gets overwritten with the address of `mpu_disable()` (recalling Listing 5.15). Consequently, the redirection of control flow to `mpu_disable()` fully disables the MPU, leaving stack memory unprotected. Now that `mpu_disable()` has been invoked, we must figure out how to execute the assembly instructions that were stored on the stack when we first injected our attack string. To do so, we need to redirect control a second time, this time to the stack address where our assembly instructions begin.

```
1  @ address          assembly code
2  @ ----------       ------------
3    @ ...instructions to disable MPU...
4    0x00000bee       bx lr
```

Listing A.3: Disassembly snippet of `mpu_disable()`

If we were to take a look at the end of `mpu_disable()`'s epilogue, given in Listing A.3, we see the last instruction branches to the address stored in LR. Recalling from Listing A, we know LR was updated by the *bl* instruction on line 7 to contain the next address in `vulnerable()`. This means that right after our attack string

caused `mpu_disable()` to be invoked, `mpu_disable()` not only disabled the MPU, it also redirected us back into `vulnerable()`. This presents us with an opportunity to redirect control flow a second time.

Once back into vulnerable, $0x38$ is added to R7, then this R7 value is set to be the SP via a `mov` instruction on line 10. Luckily we control the value of R7, meaning we can now control the SP. By adjusting the value of R7 (accounting for the addition of $0x38$) we can in turn adjust the stack values later popped off the stack. The last value popped is stored into PC, thus control flow will follow this value as before. Referring back to Listing A.2, we precisely choose R7 to point back to the beginning of our attack string. Then at the beginning of our attack string, we replace several NOP instructions with an address of one of the NOP instructions instead. As a result, the `pop` instruction on line 11 of will pop this address from SP (adjusted by our R7 value) into PC, then this value stored in PC will start execution of our assembly instructions. Given that the MPU has been disabled, these instructions successfully execute and cause the LED to blink as before.

In conclusion, we disable the MPU then get redirected back into `vulnerable()`, from there we can overwrite PC with the stack address of our assembly instructions included in the initial attack string. We are able to circumvent the MPU due to a lack security mechanisms, such as that for privilege separation, meant to prevent access to security critical registers, such as those to configure the MPU.