EXTENDING THE LIFECYCLE OF IOT DEVICES USING SELECTIVE DEACTIVATION

A Dissertation
Presented to
The Academic Faculty

By

Michael Hesse

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Computer Science

Georgia Institute of Technology

August 2020

EXTENDING THE LIFECYCLE OF IOT DEVICES USING SELECTIVE DEACTIVATION

Approved by:

Professor Taesoo Kim, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Brendan Saltaformaggio
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Mustaque Ahmad
School of Computer Science
Georgia Institute of Technology

Date Approved: April 29, 2020

ACKNOWLEDGEMENTS

I would like to thank my advisor Taesoo Kim and the other members of the committee, Brendan Saltaformaggio and Mustaque Ahmad, for their time, patience and guidance, as this thesis would not have been possible without them. I would further like to thank Meng Xu and Fan San for their continued support and feedback during the course of my research. Ranjani and Sundar, thank you for emotional support and interest in my work. Finally, I would like to thank my parents and my sister for all the love and support they continue to provide throughout my entire life.

TABLE OF CONTENTS

Acknowledgments iii

List of Tables viii

List of Figures ix

Nomenclature x

Chapter 1: Introduction 1

1.1 Problem Description 2

1.2 Goal 2

1.3 Assumptions & Threat Model 4

1.4 Security Properties 5

1.5 Challenges 6

1.6 Thesis Overview 6

Chapter 2: Background 8

2.1 Platform Architecture 8

2.2 Hardware Assisted Security Mechanisms 9

2.2.1 ARM Virtualization Extensions 9

2.2.2 ARM TrustZone 10

2.2.3 IOMMU / SMMU 12
2.3 I/O Device Virtualization Strategies 13
2.3.1 Passthrough Assignment 13
2.3.2 Full Virtualization 13
2.3.3 Paravirtualization 14
2.4 Device Tree 14
2.4.1 Device Tree Usage 16

Chapter 3: Analysis 17

3.1 What Devices To Disable 17
3.2 Device Dependencies 18
3.2.1 Parent-Child Relationship 19
3.2.2 Interrupt Dependencies 19
3.2.3 Disabled Devices in the Device Tree 20
3.3 Enforcing Deactivated Devices 21
3.3.1 Device Conflicts 22
3.3.2 Resource Conflicts 23
3.3.3 Sub-page MMIO Regions 24
3.4 Virtualization and Device Dependencies 24
3.5 Deactivating Devices as a Partitioning Problem 25
3.6 I/O Device Virtualization Strategy 26
3.6.1 Guest Managed Device Virtualization 27
3.6.2 Hypervisor Managed Device Virtualization 27

3.6.3 Shareability 28
3.7 Economy of Virtualization versus Deactivation 28
3.8 Considerations For Disabling Devices on Peripheral Buses 30

Chapter 4: Proposed Framework 32
4.1 Overview 32
4.2 Boot Sequence 34
4.3 Configuration Sources 35
4.3.1 System device tree 36
4.3.2 System configuration 36
4.3.3 Toggleable devices 37
4.3.4 Generated configurations 37
4.4 Configuration Core 38
4.4.1 Device Assignment 38
4.4.2 Generating the Guest Device Tree 38
4.4.3 Hypervisor Configuration 39
4.5 Deactivating Devices Outside the Device Tree 40

Chapter 5: Implementation And Discussion 42
5.1 Implementation 42
5.1.1 Overview 42
5.1.2 Configuration 46
5.1.3 Extracting Dependencies from the Device Tree 46
5.2 Example: Raspberry Pi 4 47

5.2.1 Example Fully Virtualized Device: Ethernet Controller 47
5.2.2 Example Virtualized Device: I2C Bus Controller 49
5.2.3 Other Raspberry Pi Components 50
5.3 Evaluation 51
5.3.1 Tamper Proof 51
5.3.2 Complete Mediation 51
5.3.3 Correctness 52
5.3.4 Availability / No Interference 54
5.4 Discussion 54
5.4.1 Limitations 54

Chapter 6: Related Work 57
6.1 Notary 57
6.2 LTZvisor 57

Chapter 7: Conclusion 58
7.1 Future Work 58

Appendix A: Device Tree and Dependencies 61
A.1 Raspberry Pi 4 Device Tree Source Code 61
A.2 Raspberry Pi 4 Device Dependency Graph 73

References 76

LIST OF TABLES

3.1 Overview of introduced virtualization classifications 31

5.1 Lines of code for each component of our implementation. 53

LIST OF FIGURES

2.1 Typical ARM single board computer block schematic. 9

2.2 ARM TrustZone architecture 11

3.1 Minimal example for a device conflict 22

3.2 Device partitions visualized in an example dependency graph 27

4.1 Proposed system architecture 32

4.2 System configuration flow 36

5.1 Jailhouse hypervisor initialization. 42

5.2 Implementation system architecture 44

5.3 Configuration for the Raspberry Pi 4 ethernet controller in the system configuration. 49

A.1 Device tree source code of the Raspberry Pi 4 72

A.2 Visualization of the Raspberry Pi 4 device tree 73

NOMENCLATURE

AMP	Asynchronous Multiprocessing
BIOS	Basic I/O System
CPU	Central Processing Unit
CSI	Camera Serial Interface
DMA	Direct Memory Access
DT	Device Tree
DTB	Device Tree Blob
DTS	Device Tree Source
EL	Execution Level
GIC	Generic Interrupt Controller
GPIO	General Purpose I/O
HAL	Hardware Abstraction Layer
I^2C	Inter-Integrated Circuit
I/O	Input/Output
IoT	Internet of Things
IOMMU	I/O Memory Management Unit
IOVA	I/O Virtual Address Space
IVSHMEM	Inter-VM Shared Memory

MDIO	Management Data I/O
MMC	MultiMedia Card
MMU	Memory Management Unit
MMIO	Memory Mapped I/O
OS	Operating System
PCI	Peripheral Component Interconnect
PCIe	PCI Express
RAM	Random Access Memory
SBC	Single Board Computer
SDIO	Secure Digital Input Output
SMMU	System Memory Management Unit
SoC	System on a Chip
SPI	Serial Peripheral Interface
SPI	Shared Peripheral Interrupt
TCB	Trusted Computing Base
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
vGIC	Virtual GIC
VM	Virtual Machine

SUMMARY

IoT devices are known for long-lived hardware and short-lived software support by the vendor, which sets the wrong security incentives for users of expensive IoT systems. In order to mitigate as many known vulnerabilities as possible after the vendor has stopped providing security patches for an IoT device, we present a framework that allows the user to selectively disable single hardware components which provide non-essential features that are associated with said vulnerabilites. In the same way, the framework can also be used proactively to reduce the attack surface of an IoT device by disabling unused features. The user's selection is enforced by a trusted computing base using different hardware security mechanisms on the ARM platform. To this end, we analyze the common hardware architecture of embedded ARM systems using the example of the Raspberry Pi 4. We conclude that only virtualization provides a fine-grained enough partition capabilities for the purpose of partitioning the hardware into used and unused components. However, we also show how other security mechanisms including IOMMUs and ARM TrustZone could be used as an optimization in some cases.

Finally, we give a proof of concept implementation using the Raspberry Pi 4 and the Sense HAT as a simulation of a complex IoT device and show how 6 of its hardware components can be selectively enabled and disabled.

CHAPTER 1
INTRODUCTION

The Internet of Things (IoT) has seen a rapid gain in popularity in recent years, both among consumers but also in industrial settings. No computing environment seems to be able to escape the pervasiveness of IoT devices. At the same time, the constantly increasing number of deployed IoT devices has produced new security and privacy challenges on a new scale. Due to the connected nature of these devices, the security implications are not limited to the device itself but can affect other parts of the internet. For example, the Mirai botnet abused more than 145,000 devices to launch a distributed denial of service (DDoS) attack with a volume of 1Tbps [1]. One of the key contributors allowing large scale attacks like the Mirai botnet is the wide-spread use of outdated software with known vulnerabilities among IoT devices. This also becomes apparent by a difference in the lifespan of the hardware and software used in IoT devices. Although the lifespan of IoT hardware is estimated by some vendors to be more than 10 years [2], as a user it is more reasonable to expect software updates, including critical security patches for only a few years. This is also confirmed by a 2020 survey of IoT devices, in which the researchers found that 83% of medical imaging devices were using software that is no longer maintained [3]. After a vendor drops software support for a device, it is merely a matter of time until vulnerabilities are discovered in device drivers, the operating system powering the IoT device or the application layer. For this reason, a long-lasting security integration in the entire life-cycle of IoT devices is integral. This has also caused governments of several countries to push for new IoT security regulations [4, 5]. For example, the State of California's Senate Bill 327 can be summarized as requiring 'reasonable security feature or features that are appropriate to the nature and function of the device' [6].

1.1 Problem Description

Specifically, for this thesis we want to focus on more expensive and powerful, long-lived IoT devices. While there is already a large share of cheaper IoT devices being operated with unmaintained software, in case of more pricey devices there are additional economic incentives to keep operating potentially vulnerable devices as opposed to buying a newer model. Typical examples would include smart fridges and other home appliances equipped with smart features. Since such vulnerability-prone IoT devices are unlikely to receive security patches after a few years, alternative methods aiming at reducing the risk and containing the threat posed by the operation of vulnerable IoT devices must be explored.

1.2 Goal

The main idea we explore in this thesis is that users could still prevent exploitation of unpatched vulnerabilities by deactivating and not using those hardware components associated with vulnerabilities. On one hand, this includes hardware components whose driver, or the kernel subsystem directly interacting with the driver, contain vulnerabilities. On the other hand, hardware components associated with vulnerabilities include those components subject to exploitation because of faulty application-level logic.

For instance, a smart fridge may be comprised of numerous components such as [7]:

- Touch display
- Cameras inside the fridge or freezer
- Compressor and other components of the cooling cycle
- Ice maker and dispenser
- Speaker, potentially microphone
- Hardware accelerators for multimedia

- External storage device interfaces such as USB or SD card readers for updating and customization purposes
- Networking interfaces including WiFi, Bluetooth and potentially ethernet

A subset of these components, in this example the last three, is directly wired on the main board, whereas other components are connected to it via some peripheral interface or bus. Although some of these components are essential for the main function of the fridge, other components, which can be on chip or some peripheral device, could be deactivated to mitigate a vulnerability. For example, if the driver for hardware accelerated multimedia decoding is vulnerable to remote code execution via an attacker controlled input, it would suffice to deactivate only the hardware acceleration chip, and thereby its driver, instead of replacing the entire fridge.

We envision this method being used in two scenarios. First, the user could proactively disable hardware components that provide features the user does not want to use in order to reduce the overall attack surface of an IoT device. Second, the user could retroactively disable hardware components which are associated with vulnerabilities to prevent exploitation or at least limit the amount of damage that attackers could cause.

The goal is therefore to entirely prevent the operating system (OS) from accessing those deactivated devices using a hardware abstraction layer (HAL) acting as a trusted computing base (TCB) for securely managing hardware components. The configuration should be controllable via a configuration interface that is part of the TCB. The TCB should then enforce the access restrictions for hardware components as defined by the user so that no software, including the operating system, can access it. Since a large share of IoT devices is powered by the ARM platform, our discussion and conclusions will be based on the ARM architecture and security mechanisms.

1.3 Assumptions & Threat Model

Given our initial goal, the work presented in this thesis is based on several assumptions for IoT devices on which the TCB may be deployed:

Remote Attacker. Attackers do not have physical access to the IoT device. Instead, they may be able to communicate with the IoT device via any networking interface from within the local network or via the internet. This includes the ability to craft arbitrary network traffic.

The OS cannot be trusted. Most operating systems are complex and large. Under the conditions given in our problem description, especially in the scenario of retroactively deactivating hardware components, we must assume that the IoT device's software might already be compromised. Since the OS could be affected as well, a self policing approach does not satisfy our security requirements (see section 1.4).

All boot programs are trusted. All bootloader stages, as well as firmware or other software executing before the first operating system is booted, only perform device initialization and do not use hardware components or perform any other actions that could be exploited by remote attackers.

Static hardware. The hardware configuration of the IoT system does not change, and we therefore have perfect and complete knowledge of all hardware components present and controllable by the operating system. This includes hardware devices which are dynamically discoverable and hot-swappable. That means, a dynamically discovered hardware component is always assigned the same address on the respective bus.

Correct user behavior. The user uses the configuration interface as intended. In scenarios where the user interacts with the configuration application via the same hardware, the user correctly identifies based on some indicator that they are using the TCB's

configuration interface as opposed to the guest mimicking the interface in order to prevent the user from disabling devices.

1.4 Security Properties

In addition to our assumptions and the functional goals, we also have several security requirements for a TCB managing hardware resources. The requirements can mostly be derived from those of a trusted computing base:

Tamper Proof The operating system or application layer cannot enable or disable devices at will. This includes the current system configuration as well as the configuration application controlling what devices will be accessible to the OS. The TCB must also make sure that the operating system cannot modify boot programs.

Complete Mediation It is impossible for the operating system or applications running on top of it to access hardware components that should be inaccessible per the user's configuration. To this end, all hardware resources must be strictly controlled by the TCB, most importantly the physical address space, including memory mapped I/O (MMIO) regions, and interrupts.

Correctness The TCB should not contain vulnerabilities or logic bugs allowing the operating system or applications to access devices. Generally, this means the TCB should be as small and contain as little complexity as possible, thereby minimizing the chance of bugs.

Availability / No Interference The operating system or applications are not able to block the TCB from operating or prevent the user from disabling a hardware component, for example by blocking hardware resources required to operate the configuration application.

1.5 Challenges

Restricting the OS access to hardware components seems like a simple goal, however its realization can be quite challenging. By far the biggest challenge is ensuring that the deactivated components can indeed not be accessed by the operating system. This essentially requires us to partition the IoT device components into those usable by the operating system and application, those controlled by the TCB and those which are disabled. As we will show in our analysis chapter, due to the internal complexity of IoT devices a hypervisor based TCB using virtualization to restrict access to selected hardware components cannot be avoided using current hardware components. This means that known input/output (I/O) device virtualization challenges have to be overcome. This is exacerbated by the fact that not all devices are accessed in the same way via a common bus. Instead, the components on a system may be spread over multiple bus systems each with their own characteristics and security properties which we have to consider. As a result, there is no single silver bullet strategy for making sure that the operating system has only access to enabled devices.

Because the hardware components must be split into the TCB and OS or application domain, it is possible that both domains require access to the same hardware components. Therefore, in order to design a IoT device using selective deactivation, the relationships between hardware components (dependencies) must also be considered.

1.6 Thesis Overview

The remainder of this thesis is structured as follows: in the next chapter we first introduce some basic concepts and background information required to better understand the challenges and our proposed framework. This includes a broad overview of the architecture of ARM based systems, security mechanisms on the ARM platform, device virtualization techniques as well as a primer on device discovery in the Linux kernel using the device tree. Specific examples are given based on the Raspberry Pi 4 not only in this chapter but

throughout this entire thesis, since it served as our development device.

In chapter 3, we analyze the facts presented in our background chapter with regard to our goals and security requirements. More specifically, we analyze dependencies in hardware components using the device tree and how they translate into requirements or restrictions with regard to devices which the user might want to disable. This analysis shows that virtualization is required to partition the hardware into multiple domains. Further, our analysis is helpful for implementing the framework for specific IoT devices in order to combine device virtualization methods, which allows to optimize the implementation for the smallest possible TCB.

This is followed by a description of our proposed framework in chapter 4. While we acknowledge the usefulness of TrustZone technology for restricting access to controllers on the system on a chip (SoC) or single board computer (SBC), we focus on using virtualization for enforcing disabled devices in this thesis. In addition, we present our implementation of the framework and discuss our case study based on the Raspberry Pi 4 in chapter 5. Finally, we give an overview of related research in chapter 6.

CHAPTER 2
BACKGROUND

In order to get a better understanding of how the stated goals can be achieved, we need to understand a number of mechanisms first. Keeping practicality in mind, we will focus on Linux systems on the ARM platform. In this chapter we describe the required background knowledge on which we will base our proposed framework, which includes the ARM system architecture, hardware assisted security mechanisms and the device tree. To this end we use the device tree of the Raspberry Pi 4 and its visualization as examples. Both are attached to this thesis in the appendix, see figures A.1 and A.2. Lastly, we give a short analysis of these concepts and how they relate to our goals.

2.1 Platform Architecture

A typical ARM hardware system architecture which we consider for this thesis is shown in figure 2.1. Here, the central processing unit (CPU) connects to other controllers residing on the SoC using a system interconnect, which is usually based on the Advanced Microcontroller Bus Architecture (AMBA) on ARM systems. Additional peripheral controllers can be added on the board hosting the SoC. These peripherals are connected to the peripheral interconnect, a separate bus for controllers on the SBC utside the SoC, which is connected to the system interconnect via a bridge controller. From the CPU's point of view, as well as that of other board components, the partition into the system and peripheral interconnect is hidden and requires no extra software configuration. Next to the generic interconnects, there can also be direct connections between controllers. External peripherals and circuits may be connected to the system via any peripheral interface supported by one of the controllers on the board. For example, the Raspberry Pi 4 can connect to external peripherals using the Inter-Integrated Circuit (I^2C) bus, Serial Peripheral Interface (SPI) or universal

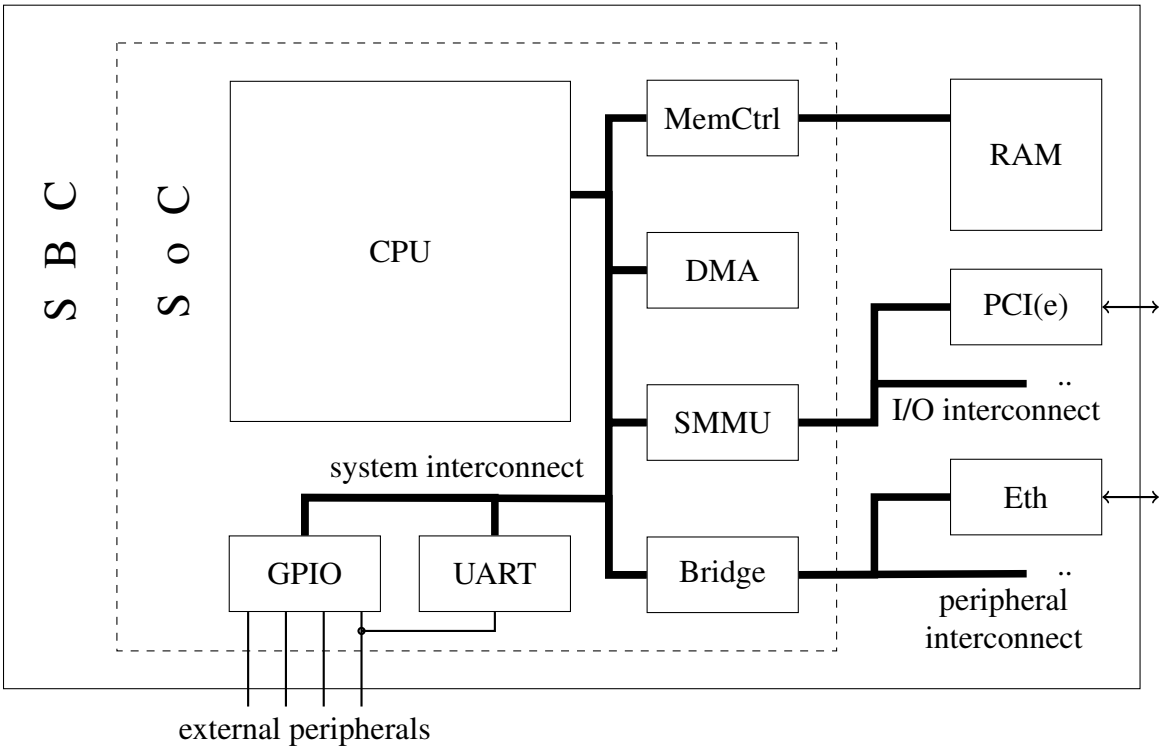


Figure 2.1: Typical ARM single board computer block schematic.

asynchronous receiver/transmitter (UART) on general purpose I/O (GPIO) pins. For our purposes of enabling or disabling devices, we will consider all controllers on the SoC, the rest of the SBC and external systems connected via one of the peripheral controllers.

2.2 Hardware Assisted Security Mechanisms

2.2.1 ARM Virtualization Extensions

ARM's hardware virtualization support provides separation and virtualization mechanisms for both memory and interrupts [8]. Switching execution between different virtual machines (VMs) is supported by an additional execution mode, execution level (EL) 2 or hyp mode, in which the hypervisor executes. Similar to hardware virtualization on other platforms, the memory management unit (MMU) must support a second address translation stage in order to translate from guest physical addresses to machine physical addresses. By selecting which

address ranges are mapped into the guest physical address space, the MMU can control which memory regions and devices the VM can access. This allows memory partition with page granularity, but even more fine-grained access restrictions for VMs can be achieved using the trap and emulate method.

Further, interrupts are masked and virtualized using the Generic Interrupt Controller (GIC), which provides two separate interfaces for CPUs and virtual CPUs [9]. The hypervisor only has to assign and mask a virtual GIC (vGIC) to each virtual machine once. Then, each VM can directly handle interrupts without any need for emulation by the hypervisor.

2.2.2 ARM TrustZone

ARM TrustZone is a trusted execution environment for the ARM platform which divides the system into two worlds, a secure and a non-secure world [10, 11]. The more powerful TrustZone variant designed for the ARM v8-A instruction set architecture uses an additional privileged execution mode, EL3, to manage the two worlds and world switches in the secure monitor. This means that TrustZone operates orthogonally to existing hardware virtualization support and does not preclude usage of the latter, as show in figure 2.2. However, hypervisors in the secure world can only be used as of ARM v8.4-A and newer [9]. The isolation of the two worlds is achieved in hardware by introducing an additional address bit on the system interconnect, indicating which world a transaction is part of. As a result, controllers connected to the system or peripheral interconnect are also partitioned into the two worlds, with the option of sharing a controller between both worlds if supported by the controller. Further, TrustZone also facilitates the GIC and vGIC interfaces to deliver interrupts to both worlds separately without software-side emulation. The partition into the two TrustZone worlds is only possible on a per-device granularity.

In addition to its intended use as a trusted execution environment, TrustZone can even be used for virtualization purposes in different setups as shown by Pinto and Santos [11].

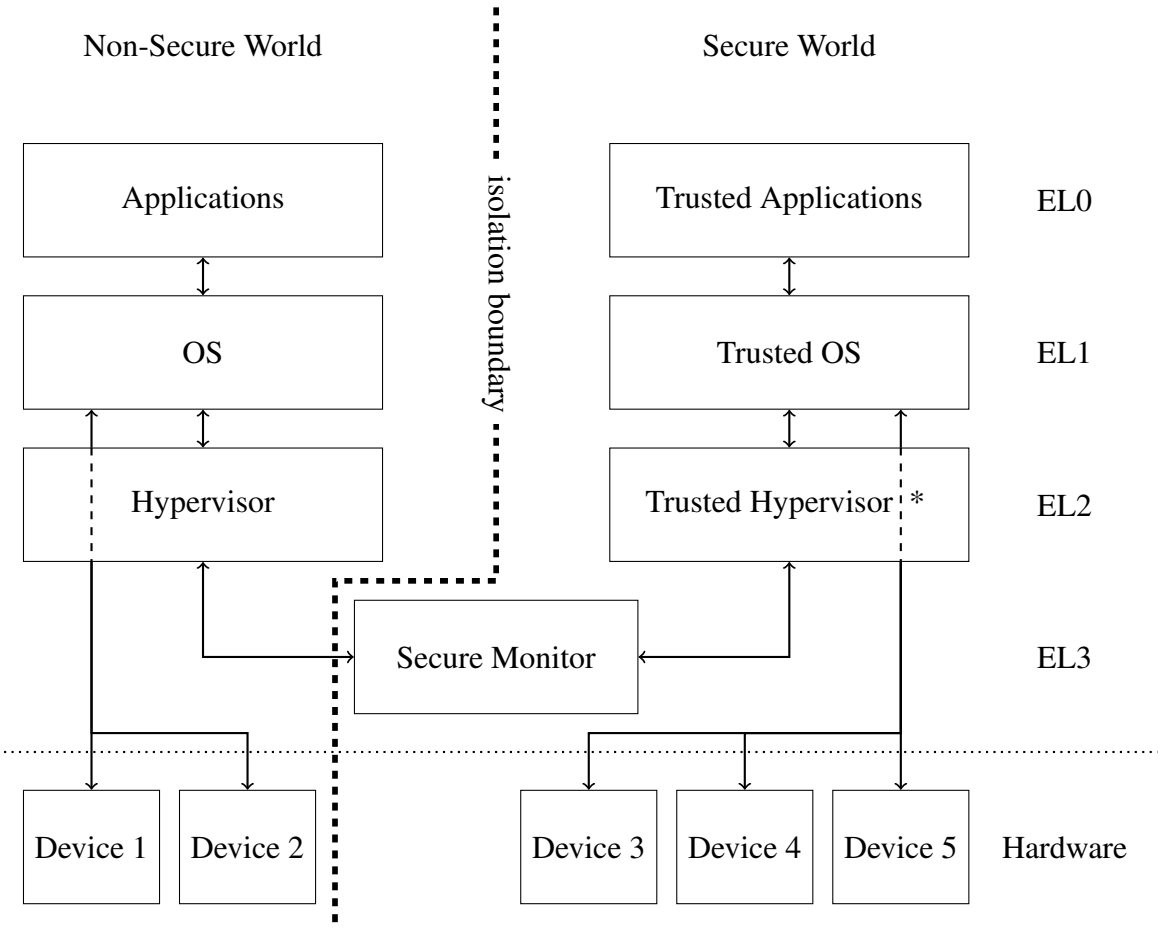


Figure 2.2: ARM TrustZone architecture [8, 9, 10]. A trusted hypervisor is only possible in ARM v8.4-A and newer.

2.2.3 IOMMU / SMMU

The I/O memory management unit (IOMMU), called System Memory Management Unit (SMMU) on ARM, is the analog to the MMU for I/O devices. It allows the hypervisor to give a VM full control of a peripheral controller and its MMIO registers [12], even in the absence of TrustZone. In addition, with an SMMU the partitioning guarantees given by TrustZone or virtualization can be extended to peripheral devices thar are neither connected to the system nor the peripheral interconnect.

Although a passthrough assignment to a virtual machine is possible without an SMMU if the device's registers are identity-mapped into the guest physical space (guest physical address = machine physical address), this setup is susceptible to direct memory access (DMA) attacks [13]. In this scenario, a rouge VM could abuse the device's DMA capabilities to read or even manipulate memory regions which are not mapped into the VM's address space. This means the VM could manipulate the hypervisor's memory or interact with registers of devices which should be inaccessible.

As shown in figure 2.1, the SMMU mediates between the system interconnect and client devices, in this example a single Peripheral Component Interconnect (PCI) controller. Controlled by the hypervisor, the SMMU performs the second stage address translation from the guest physical address space to machine physical address space for DMA transactions, which means that VMs can now directly control client I/O devices connected via an SMMU [12]. Similar to the configuration of the MMU, the hypervisor can control which memory regions can be accessed by clients by mapping memory regions into a separate I/O virtual address space (IOVA). In the case of older SMMU models, one SMMU is required for each client, but newer models support multiple devices at a time. Since each transaction from a client device is tagged with a unique identifier, clients can be operated in separate IOVAs for isolation purposes, as well as in joint IOVAs to facilitate communication within subsets of the client devices.

2.3 I/O Device Virtualization Strategies

2.3.1 Passthrough Assignment

With the passthrough assignment, a virtual machine obtains direct control of a passed-through device, including its memory-mapped registers and its interrupts. This method introduces no complexity whatsoever, other than correct configuration of the MMU and the interrupt controller. The VM gains complete controle over the device. As we have already mentioned in section 2.2.3, a passthrough assignment of a device that can initiate DMA transactions allows a rogue VM to launch DMA attacks, which could allow it to bypass virtualization restrictions and manipulate the hypervisor.

2.3.2 Full Virtualization

A fully virtualized device is emulated for the guest at the register level. The advantage of this approach is that the VM's operating system does not need to be modified in order to be able to operate the emulated device.

A typical implementation for a full virtualization driver of a physical device is the trap and emulate approach, which simply forwards all register accesses to the physical device, thereby acting as a proxy for each register access by VM. For example, this method can be used as an alternative to an `SMMU` in order to prevent DMA attacks for devices with simple register interfaces. In case of the ethernet controller in the Raspberry Pi 4, DMA transactions are controlled by the driver by writing the buffer address and size two device registers before the transaction is started [1]. Here, access to DMA address registers could be mediated by the hypervisor to prevent otherwise illegal memory accesses.

[1] See the ethernet driver `/drivers/net/ethernet/broadcom/genet/bcmgenet.c` in the Linux kernel [14]

2.3.3 Paravirtualization

Paravirtualization emulates a physical device on top of the HAL interface, thereby abstracting away the characteristics of the concrete device that is emulated. However, the OS running in the virtual machine requires a special paravirtualization driver, called the frontend driver, in order to interact with its paravirtualization counterpart in the hypervisor, the backend driver. For this thesis, we will consider paravirtualization drivers to be implemented on top of the hypervisor's hardware abstraction layer.

Compared to full virtualization, the advantage of paravirtualization is the possibility of reusing the paravirtualization driver, since it only needs to be implemented once for an entire class of devices, as opposed to full virtualization which must be reimplemented for each concrete device.

2.4 Device Tree

Unlike the x86 architecture, where the Advanced Configuration and Power Interface (ACPI) is the industry standard for dynamic device discovery at runtime, the most common method for enumerating hardware devices in an ARM-based embedded system is the device tree (DT).

The device tree is a static description of a system's hardware components and resources, as well as how they relate to each other. As the name suggests, the tree represents each hardware component as a node in a hierarchical tree structure (see the DT for the Raspberry Pi 4 as an example in figure A.1 in the appendix). Each node in the tree stands for:

- a **bus** or **bus controller**, which is usually a branch node. Buses providing their own device discovery or probing mechanism, like PCI controllers, can be leaf nodes. Each bus creates its own address space in which its child nodes can be addressed. `simple-bus` nodes are special platform internal I/O buses that do not support device probing and do not require any other configuration or software support in order

to access the devices represented by their child nodes.

- a **hardware chip**, mostly leaf nodes in the tree, providing a certain functionality.
- a more **fine-grained resource**, such as a single GPIO pin. This can also include somewhat abstract resources such as a power domain node, which describes in what groups hardware components can be powered on or off.

The root node is a special node representing the physical address space.

Further, each node contains key-value properties describing the device that is represented. Internally represented by a bytestring, on a higher level all values can consist of a combination of unsigned 32 or 64 bit integers, strings, bytestrings or references to other nodes. There exists a set of properties specified in the DT specification [15], however all other properties have device-specific interpretation and meaning. The most important properties include:

- `compatible`, a list of strings each identifying a compatible device driver.
- `reg`, a list of regions in the parent's address space where the device's registers are mapped.
- `interrupts`, a list of interrupt specifiers describing which interrupts the device may raise. In the simplest case, the specifier is the hardware interrupt number.
- `ranges`, which maps portions of the address space created by a bus into its parent's address space, effectively making it accessible to every other device on or with access to the parent's bus.
- `phandle`, a unique numerical identifier meant for referencing a node. For further description of this feature, see the analysis on device dependencies in section 3.2.

Since the root node represents the physical address space on a system, this means that each of the root node's child nodes with a `reg` property represents a MMIO device. Using

the `ranges` property, even devices which are not direct children of the root node can be memory mapped. For example, all child nodes of `/soc` and `/scb` in the Raspberry Pi 4 device tree (see figure A.1) are also memory mapped if they have a `reg` property.

Properties can also contain references to other nodes in order to describe how devices interact with each other. A node is uniquely identifiable via its path in the device tree or using its `phandle`.

2.4.1 Device Tree Usage

Once compiled from its source code representation as shown in figure A.1, which is called the device tree source (DTS), the resulting device tree blob (DTB) is installed on the boot medium to make it accessible to the system's firmware, bootloader and hypervisor [15]. During the compilation, all properties are packed from their higher level representation (strings, bytestrings, unsigned integers, references) into a byte sequence of variable length.

At each stage in the boot process, the boot program can initialize hardware devices and make changes to the device tree as required, including adding, modifying or removing nodes. For example, the firmware of the Raspberry Pi 4 adds a property to the ethernet controller describing the controller's MAC address. The modified device tree is then transferred to the next stage in the boot process, along with the control flow. At last, a reference to the device tree is passed to Linux when it is started, which then uses the DTB as the single source of truth about the hardware components on the system and uses it to determine and load the appropriate driver for each device. In turn, each device driver instance uses the properties of the representing node for its own configuration. Device nodes may also be added or reconfigured at runtime by applying overlays on top of the tree. Further, bus controllers with integrated probing or device discovery functionality may dynamically load device drivers which will not be accounted for in the device tree.

CHAPTER 3
ANALYSIS

Based on the concepts and systems described in the previous chapter, we further clarify the consequences of our goals and security requirements in this section. Further, we introduce new concepts we deem to be helpful in better understanding of the relationships between hardware components. This analysis is equally valuable for our framework design and later on for the implementation of the framework for a particular IoT device in order to make the right design decisions for the smallest possible TCB.

3.1 What Devices To Disable

At first, it is important to specify the hardware components for which deactivation makes sense. Reusing our example from the introduction, this may include both on-chip as well as external peripheral components.

Therefore, it is desirable to be able to deactivate all peripheral interfaces connected to external peripherals that are non-essential to the IoT system's functionality. In addition, some interfaces allow multiple peripherals to be connected to the same peripheral controller, for example PCI, I^2C or SPI. In these cases, blocking access to a single connected peripheral, that is, its address on the peripheral bus, is preferred over disabling the entire peripheral controller. Not all peripherals connected via a peripheral controller are manifested in the device tree, since the device tree only contains devices that are controlled by a kernel driver. As a result, peripheral controllers or their child nodes are usually leaf nodes in the device tree.

In addition, some hardware components on the SBC may be candidates for deactivation as well, as long as they provide non-essential functions. For example, the multimedia accelerators or WiFi chip components could be considered on-chip components for which

deactivation is desired. Again, these are typically leaf nodes in the device tree. Even more, the dependency analysis in the following section will show that these devices do not even have incoming dependency edges in the dependency graph. As a matter of fact, the WiFi controller in the Raspberry Pi 4 is not even part of the device tree because it is connected via the Secure Digital Input Output (SDIO) bus controlled by `/soc/mmcnr@7e300000`[1].

For the remainder of this thesis, we will consider all components fulfilling either of these two criteria to be devices for which deactivation should be supported.

3.2 Device Dependencies

Most components in a system will no be operable on their own, but require the services provided by other controllers in order to be usable by software. In case of our example device tree of the Raspberry Pi 4, the MultiMedia Card (MMC) controller `/soc/mmc@7e300000` relies on the DMA controller `/soc/dma@7e00700` in order to read the contents from the inserted micro SD card into a memory buffer or vice versa. For every hardware component that we want to use, we need to make sure that all other devices required by such relationships are also enabled and usable. Or, put in other words, we need to perform a dependency analysis on the device tree to satisfy the dependencies of all enabled devices.

For this thesis, we will consider each reference in the device tree to be a dependency of the referencing node, called consumer, on the referenced node, called provider. Some references also include a specifier which details exactly which resource is consumed by the device. This is required for providers which manage multiple resources of the same kind, such as interrupt controllers, GPIO controllers and clock providers. For instance, the device `/soc/i2s@7e203000` consumes the output clock `BCM2835_CLOCK_PCM` (the macro resolves to clock number 31) of the clock provider `cprman`, as well as DMA channels 2 and 3 of the SoC's DMA controller. In addition to dependencies mentioned explicitly in

[1]This can be verified by running the following two commands on a standard installation of Raspbian on the Raspberry Pi 4 (`brcmfmac` it the name of the WiFi driver):

```
ls -la /sys/class/mmc_host/mmc1/
ls -la /sys/class/mmc_host/mmc1/mmc1\:0001/mmc1\:0001\:{1,2}/
```

properties, we also consider two other sources of dependencies.

3.2.1 Parent-Child Relationship

Firstly, each device in the device tree depends on its parent node, since the OS can only communicate with devices via the controller of the bus on which said child is addressed. One exception is the root node representing physical address space, which is directly accessible by the CPU. Also, some bus nodes do not represent an actual bus controller and map the entire address space created by the latter into their parent address space. For example, the `/soc` and `/scb` nodes in the device tree of the Raspberry Pi 4 (see figure A.1), such as all other buses with compatible string `simple-bus`, represent platform internal I/O buses on which child nodes can be accessed without any additional configuration. In these cases, the child-parent dependency can be disregarded, however there is no automated method known to the author to reliably determine whether or not a bus controller is required in order to access its child devices in the device tree. This information would therefore need to be encoded in the device tree.

3.2.2 Interrupt Dependencies

Secondly, each device depends on the interrupt controller serving any of its raised interrupt requests. Even though interrupts are declared as specifiers, they do not include an explicit reference to the handling interrupt controller. Instead, interrupt controllers and consuming devices are organized in interrupt domains, which can be thought of a separate tree that is orthogonal to the device tree. In order to find the interrupt controller serving an interrupt, we need to follow the `interrupt-parent` and `interrupt-map` properties in the device tree, or the parent node if no such property exists, until we reach a node representing an interrupt controller. To give an example, the I^2C controller `/soc/i2c@7e804000` in the Raspberry Pi 4 device tree can raise the shared peripheral interrupt (SPI) 117, which will be served by the interrupt controller `/soc/gicv400@40041000` because it is the root

node's parent interrupt controller.

Using these dependency relationships between devices, we can also build a dependency graph as a better representation or visualization of all relationships. Note, that in most cases the device tree will turn into a directed acyclic dependency graph and in some cases even into a graph containing cycles. As an example, we visualized the dependency graph of the Raspberry Pi 4 in figure A.2 in the appendix. The ethernet controller in the lower right corner loops back to itself via a dependency on a device on its child Management Data I/O (MDIO) bus (see figure A.2 in the appendix). This graph also shows the central role of a few select devices which are providers for most other devices in the device tree: the interrupt controller, the GPIO chip via its child nodes representing single pins or pin groups, and the main hardware clock provider `cprman`.

In general, we need to assume that a dependency is mandatory because we do not know whether the consumer's driver works without the provider's services. Some drivers might have a fallback option, or the dependency might provide optional functionality, however we cannot make this assumption in the general case. In practice this means that in order to be able to use a certain hardware chip, we also need to make sure that we are operating all of its providers and their drivers. This becomes important during the initial design phase of a system.

3.2.3 Disabled Devices in the Device Tree

We also want to discuss the relationship of the devices, which we expect to be candidates for deactivation, with the device tree. These hardware components typically provide functionality that can directly be interacted with or seen by the user of the IoT system. In addition, they are mostly leaf nodes in the dependency graph with no consumers other than child nodes. The only exceptions are peripheral bus interfaces, which means the devices connected to that peripheral bus are dependent on the bus controller as a parent-child dependency. This theory can be confirmed by looking at device trees of typical IoT boards such as the Raspberry Pi 4

in figure A.2. All the devices that we aim to make deactivatable, including I^2C, SPI, Camera Serial Interface (CSI) and UART controllers or their child nodes in the device tree, do not have any non-child dependencies. This also makes sense from a user's perspective, who sees the entire system as a black box providing some features which can be used independently, without having to worry about intra-device dependencies.

In conclusion, we can assume that all devices which our system should be able to deactivate do not act as provider for any other device that is not deactivatable. Further, any dependencies between deactivatable devices do not create cycles among them, since they are usually parent-child dependency edges. Thereby, the set of devices controlled by the TCB can remain constant regardless of which peripherals the user has enabled or disabled.

3.3 Enforcing Deactivated Devices

The IoT devices we are considering in our problem statement provide many different features, which are typically implemented across multiple integrated circuits and peripherals external to the main board. In conclusion, these components must communicate with the driving software via peripheral interfaces on the main board hosting the CPU. Reusing our initial example, many of the candidate devices for disabling in order to mitigate vulnerabilities are peripherals that are connected via some peripheral (bus) interface such as UART, I^2C, SPI or CSI. It is therefore undesirable to disable an entire peripheral bus interface as a whole if there is more than one device connected to it, since we want to retain as many features as possible while disabling the smallest number of hardware components to make sure the vulnerabilities in question cannot be exploited. In conclusion, only using the isolation mechanisms provided by TrustZone will not suffice for our goals. Instead, our main tool for denying the OS access to devices must include virtualization and device emulation, supplemented by the use of SMMUs wherever possible.

3.3.1 Device Conflicts

Another reason why virtualization is required is the central role played by a few controllers, which leads to device conflicts. As we have discovered in our analysis of the device dependencies in the Raspberry Pi 4, there are some core components with many consumer controllers, such as the GPIO or interrupt controller. Figure 3.1 shows a minimal example for this situation.

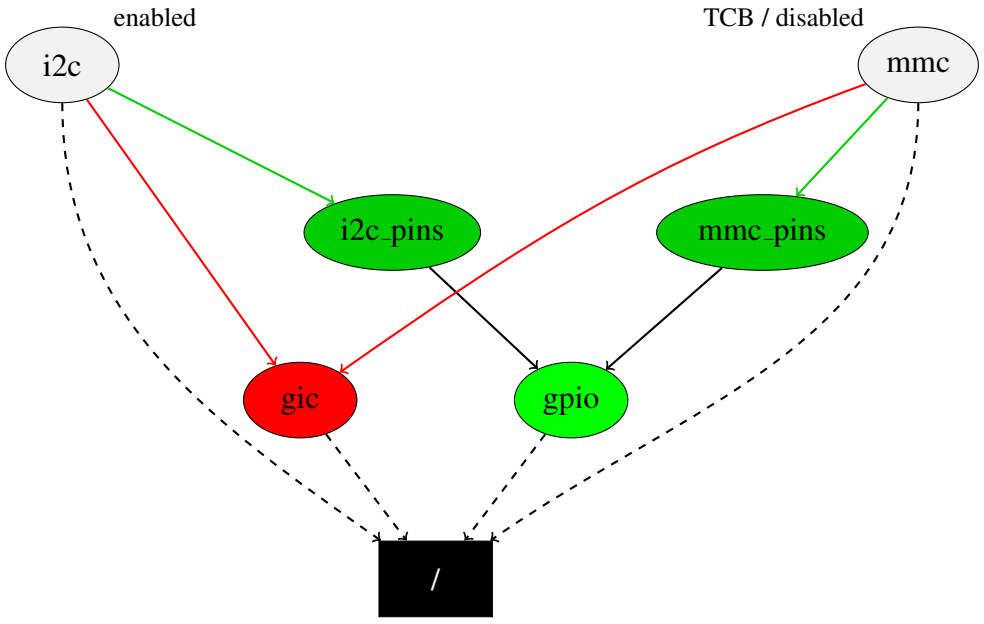


Figure 3.1: Minimal example for a device conflict. For completeness, the original parent-child relationships are indicated with dashed black lines where they are not regarded as dependencies. If the I^2C controller is enabled to be usable by the OS, and the MMC reader is security critical for the TCB, both domains would require control of the GPIO controller. The GIC allows provides separate controls via its GIC and vGIC interfaces.

This means that some of these devices' consumers will be enabled for use by the operating system, some might be disabled and some others are required by the TCB or configuration interface. Both the main OS, as well as the TCB, must be able to control some of the resources provided by the device. We call this situation, in which two devices in different domains have a common dependency, a device conflict. Unless the hardware provides special support for sharing the resources by these devices, this can only be achieved

by giving the TCB full control over the device, which has to emulate it for the guest such that permitted interactions with the device are forwarded to it. For example, let the I^2C controller in figure 3.1 be a device that is enabled for the main IoT operating system, while the MMC controller is critical for the TCB. Although the GIC provides separate interfaces for the secure world/hypervisor and non-secure world/guest, the GPIO controller can only be controlled by one domain at a time. Such a partition on a sub-controller granularity is not possible purely using TrustZone's division into the secure and non-secure world, hence we need virtualization and device emulation as an enforcement method.

Further, even if the conflicting device that is critical for the TCB is not actively being used by it, its consumed resources must be protected from the OS. For example, granting the OS full access to a GPIO controller that is the center of a device conflict would still allow the OS to use software for simulating the actions of the consuming controllers on the used pins, which would be the MMC controller in figure 3.1. This is also referred to as *bit banging*.

At the same time, the fact that we need virtualization as an enforcement method also means a shift in paradigm: instead of thinking about how we can disable and forbid access to a device, we now need to think about how devices can be made available to the virtual machine while still fulfilling our initial goals. On the upside, any device that the virtual machine is granted access to can also be disabled by simply not granting access to it.

3.3.2 Resource Conflicts

Some devices manage multiple resources which can be requested or controlled by consumers one at a time. For example, the GPIO controller manages multiple pins, and clock providers can have an arbitrary number of output clocks. Two devices are said to have a resource conflict if they consume the same resource provided by the same device. For example, two separate controllers could require the same output clock of the same provider, or the same GPIO pin. Resource conflicts can also include overlapping register regions.

Although discouraged by the device tree compiler via a warning message, we discovered several examples of devices which list registers at the same physical address. For example, the Nvidia Jetson TK1[2] contains two nodes mapping device registers in a 16kB region starting at `0x7d000000` in the physical address space, `/usb@7d000000` and `/usb-phy@7d000000`.

Conflicting consumers must always be controlled by the same domain, because sharing a resource between the untrusted OS and the TCB jeopardizes the secure operation of the consumer in the TCB. However, an enabled device controlled by the OS may have a resource conflict with a deactivated device.

3.3.3 Sub-page MMIO Regions

We also noticed that many device's registers are mapped in memory regions that are smaller than a page. In addition, we also discovered multiple systems, including the Raspberry Pi 4, which map multiple MMIO devices on the same page in the physical address space. If a disabled device is mapped to the same page as an enabled device, this configuration would only be enforceable with the trap and emulate approach using virtualization. IOMMUs would not suffice in this situation either, because they map entire pages, just like the MMU.

3.4 Virtualization and Device Dependencies

Since we will require virtualization to enforce disabled devices, we will have two domains into which devices will be partitioned. The TCB must be in complete control of all controllers and resources that are critical for it in order to enforce the virtual machine boundary, plus the hardware required so the user can interact with the configuration application. This includes the CPU, memory, interrupt controllers, all SMMUs and the block device used for loading the hypervisor. It is important to note that controlling those devices does not necessarily mean that the TCB is actively using these devices. Instead, it is sufficient to

[2] See the DTS file `/arch/arm/boot/dts/tegra124.dtsi` in the Linux kernel v4.19 [16]

block the guest from using these devices, in the same way disabled devices are shielded from the guest.

Similarly, each component might require access to its dependencies in order to provide its functionality. Because we do not trust the main OS, and by extension therefore the entire guest domain, we must make sure that all devices on which a TCB controlled device depends are also in control of the TCB. In the dependency graph, this means that all devices reachable from devices that are critical for the TCB's operation must be under control of the TCB. For our purposes, where the device assignment to different domains never changes, it is sufficient to perform this analysis once during the design phase of the system.

Vice versa, the dependencies of any device that is enabled and directly controlled by the guest domain must also be assigned to the guest, unless it is an emulated device.

3.5 Deactivating Devices as a Partitioning Problem

One of the simplest solution for splitting the hardware components into TCB and VM controlled devices would be passing all devices through to the VM, however for DMA capable devices this would violate our security requirements. Similarly, paravirtualizing all devices that should be usable by the guest also defeats our original motivation, since the device drivers would be part of the TCB, which contradicts our goal of a small TCB. Instead, the smalled possible TCB can be achieved by using different device virtualization methods for each hardware component based on each component's properties. With help of the dependency graph, we can ensure that each controller can still properly provide its functionality.

Using the device dependency graph and its requirements, we can now think of our challenges as a (graph) partitioning problem, since our goal is a partition of hardware components into enabled and disabled devices. Namely, the hardware components must be split into three domains: security critical devices for TCB, disabled devices and enabled devices usable by the main IoT OS and application. A small example is shown in figure 3.2. Every

dependency edge crossing the partition boundary requires the provider of that dependency to be paravirtualized so that the OS can access its services and resources.

In many scenarios there is only one possible way to do so, as in this example with the `spi` controller if it was enabled. However, for some configurations there can be multiple options as shown in figure 3.2, where there are two levels of nodes between a device which we want to enable or disable and the root node. In this example, the OS could control the I^2C device and virtualize its services to the operating system so that the OS can access its child devices. Alternatively, the `gpio` controller and `clock` provider could be virtualized by the TCB such that their resources consumed by the TCB are separated and inaccessible to the OS. The I^2C controller could then be passed through to the guest. Note, that the two options provide different levels of granularity for deactivating devices, because the second option would only allow the entire I^2C bus to be deactivated. On the other hand, the first option should result in a smaller TCB since only one controller must be virtualized. It is therefore crucial to define what devices should be deactivatable by the user, as we have done in the beginning of this chapter, in order to find the best partition that satisfies our requirements as well as our initial security goals.

3.6 I/O Device Virtualization Strategy

In light of our dependency analysis, it is also helpful to use another classification for the I/O device virtualization strategies presented in section 2.3. As opposed to the traditional classification that focuses on the interface exposed by the hypervisor to a VM, our classification focuses on which domain has to manage a device's dependencies, which means there are only two classes. Table 3.1 gives an overview of the classifications and virtualization methods.

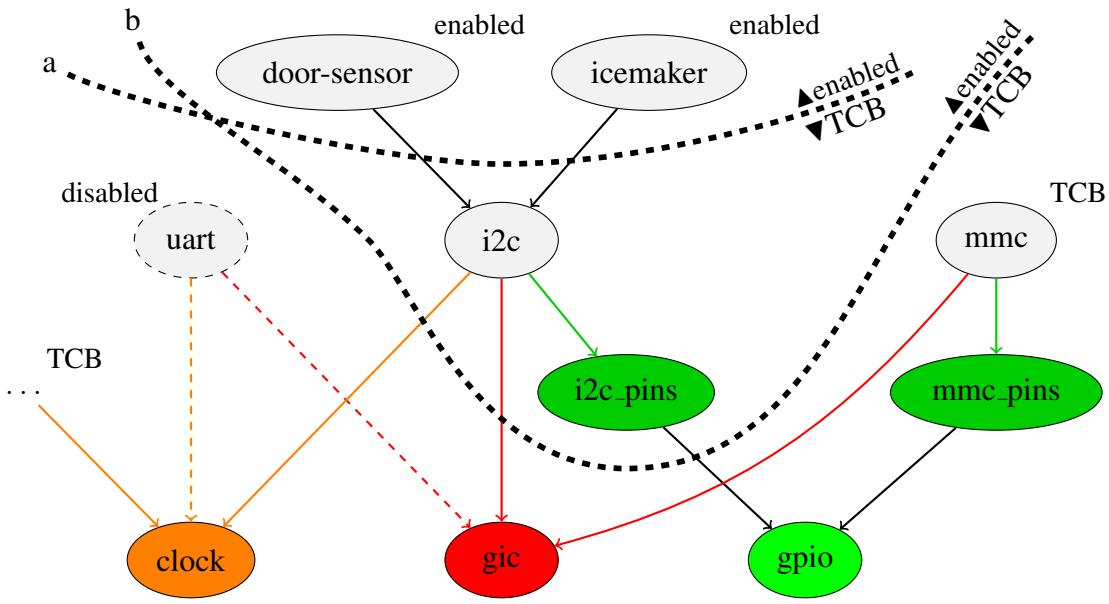


Figure 3.2: Example device partitions visualized in a graph. The label next to each peripheral indicates what domain they should be assigned to. For the sake of partitioning, the disabled devices can be thought of being controlled the TCB to reduce the number of partitions to two. However, disabled devices are not being used by the TCB in reality.

3.6.1 Guest Managed Device Virtualization

In the first case, the VM has to manage the dependencies of a virtualized device. The obvious example includes devices that are passed through to the guest. In addition, fully virtualized devices can also be considered to part of this class, since the hypervisor would normally just validate and forward register accesses to the physical registers of the virtualized device.

Since the virtual machine has to manage the devices dependencies, it must also be able to access all of the devices dependencies. Further, guest managed device virtualization adds little to no complexity to the hypervisor, making this class of virtualization drivers the preferred one.

3.6.2 Hypervisor Managed Device Virtualization

The second option is to give the hypervisor complete control over a virtualized device, including the responsibility of managing its dependencies. Only paravirtualized devices are

part of this class since the reason for a HAL is to abstract away the hardware, including dependencies between single components.

The burden of abstracting the hardare lies with the hypervisor for this class. This means that using this kind of virtualization adds the complexity of the device driver, the operating system's subsystem for the device type and the paravirtualization driver to the hypervisor. On the upside, it allows a very fine-grained control over hardware resources. In terms of dependency management, all providers for hypervisor managed devices are required to be controlled and accessible by the hypervisor.

3.6.3 Shareability

A device is called shareable or shared if it can have consumers managed by the hypervisor and guest at the same time. It is therefore not a quality of the device itself, but of a device and a virtualization driver for it. Most importantly, this is only possible if the resources or services provided by a device are strictly separate and independently usable between hypervisor and guest side consumers, since otherwise the security of the resources consumed by devices controlled by the hypervisor cannot be guaranteed in terms of integrity and availability. Shareable devices are necessary in some situations order to resolve device conflicts. This can be the case, for example, when a paravirtualized device depends on another device, which also requires paravirtualization. If a device requires no other configuration or dependencies, full virtualization can also be used for this purpose by limiting the VM's access to registers associated with the resources that should be accessible to the guest.

Per definition, the GIC is always shareable.

3.7 Economy of Virtualization versus Deactivation

It is also important to consider the effect of using I/O device virtualization as a means to later disable hardware components with regard to our initial goal of preventing exploitation of certain vulnerabilities or reducing the attack surface. This goal can only be reached if the

size and complexity of the TCB is as small as possible. Therefore, a balance between the utility of virtualizing device components and the thereby reduced complexity of the entire system must be found, not just for the state of the entire system where devices are actually deactivated but also in the standard configuration of the system where all components are activated and (ab)usable. The cost of virtualizing a hardware component in terms of added complexity to the TCB should always be reasonably low compared to the complexity of the devices which we can deactivate as a result of using the virtualized device.

Therefore, a device that should be deactivatable by the user must either be passed through to the guest or fully virtualized using a proxy-like implementation as described in section 2.3.2. Paravirtualizing a device that should be deactivatable would just shift the driver from the virtual machine into the hypervisor. This would be at odds with our understanding of a trusted TCB since the untrusted driver, which we wanted to deactivate in case of a vulnerability, would become part of the TCB.

Instead, paravirtualization proves useful for devices providing resources and services to other hardware components as an enabler for deactivation of its consumers. This includes controllers of peripheral buses, which can be understood as providing its address space as a resource to each device connected via the peripheral bus. Apart from that, paravirtualization can also be used at reasonable cost for the core components of an IoT system with many consumers, which are already required by the TCB regardless of the guest's configuration. This is the case because the hypervisor already contains a driver for these components. In the example of the Raspberry Pi 4, such devices include the GPIO controller `/soc/gpio@7e200000`, the hardware clock provider `/soc/cprman@7e101000` and the DMA controller on the SoC, `/soc/dma@7e007000`, because all of them are required by the hypervisor in order to read from the main SD card using the MMC controller `/soc/emmc2@7e340000`.

3.8 Considerations For Disabling Devices on Peripheral Buses

While there are hardware assisted security mechanisms providing security guarantees for the system and peripheral interconnect, the same cannot be said for peripheral bus systems. Each peripheral bus communication interface requires special considerations when thinking about how to enable or disable access to bus devices in a virtualized system.

For example, devices connected via PCI support memory mapped I/O and can initiate DMA transactions via the PCI controller. IOMMUs can used to mediate DMA transactions, which allows PCI devices to be passed through to virtual machine guests. The PCI controller must be fully or paravirtualized if access to some devices on the PCI bus should be restricted.

To give another example, the I^2C bus allows communication with I^2C connected devices via the bus controller [17]. For I^2C controllers supporting slave mode, other devices connected to the I^2C bus can request to become the bus master, which allows them to initiate I^2C transactions independently of the controller and thereby directly communicate with devices on the I^2C bus. It is therefore sufficient to emulate the I^2C controller and prevent usage of slave mode as well as transactions to addresses of blocked devices. Devices connected via the I^2C bus can be passed through to guests.

Table 3.1: Overview of introduced virtualization. Providers of a virtualized device must always be accessible from the managing domain.

Method	**Managing domain**	**Shareable**	**Control granularity**	**Complexity**	**Providers of device**	**Consumers of device**
Passthrough	VM	No	Device	None	VM	VM
Full Virtualization	VM	No	Device / Sub-device[1]	Low[1]	VM	VM
	-[1,2]	Yes[1,2]	Device / Sub-device[1]	Low[1]	-[1,2]	Hypervisor[1,2], VM
Paravirtualization	Hypervisor	Yes[1]	Sub-device	Medium[3]	Hypervisor	Hypervisor, VM

[1] Depends on complexity of device interface
[2] Only if the device has no dependencies
[3] Depends on complexity of device interface and the HAL interface for this device

CHAPTER 4

PROPOSED FRAMEWORK

4.1 Overview

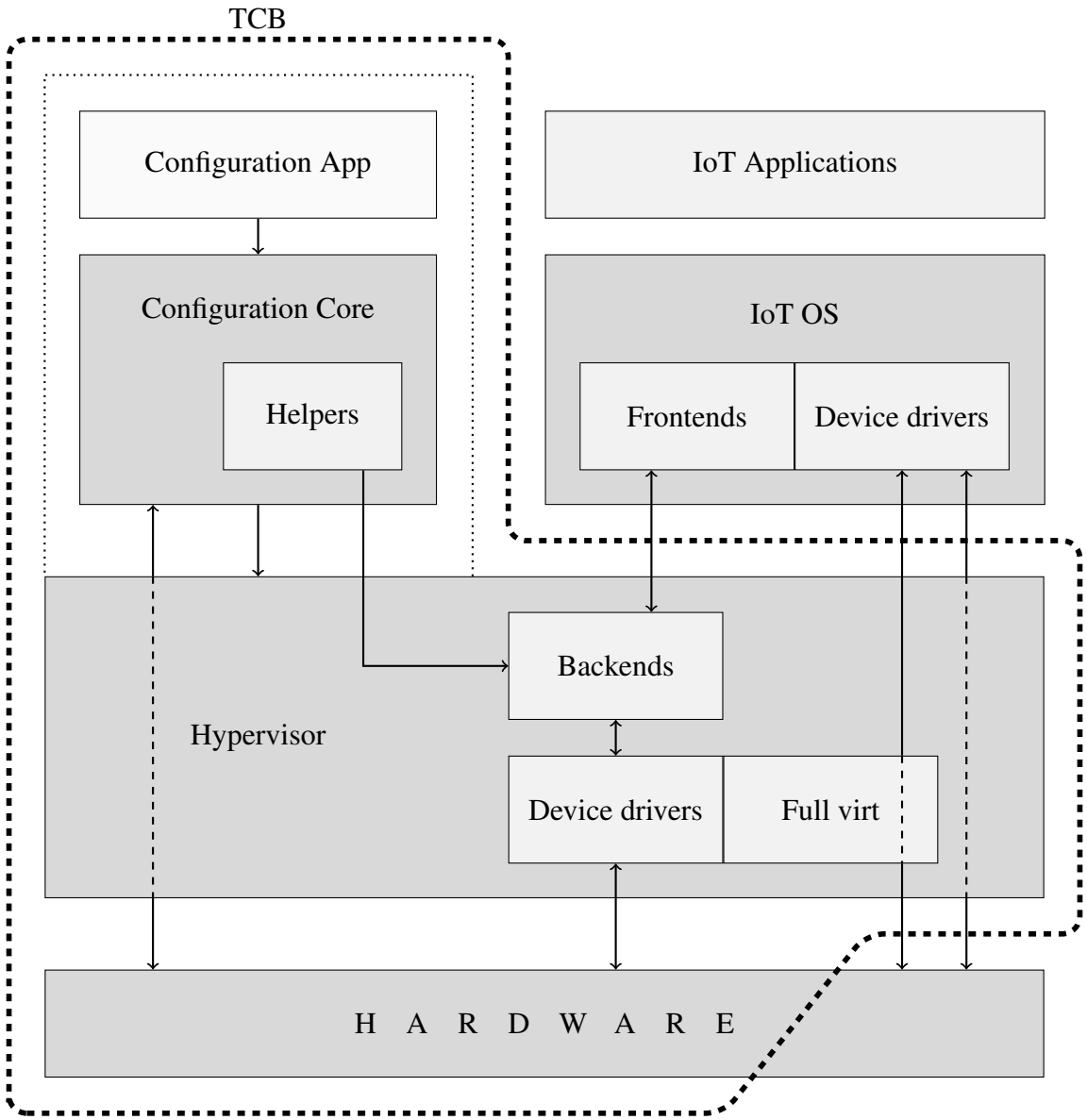


Figure 4.1: Proposed system architecture allowing single devices to be disabled using virtualization.

Based on our conclusions and requirements of our analysis in chapter 3, we developed the architecture shown in figure 4.1. The system uses a Type-1 hypervisor setup. The guest virtual machine runs the main operating system and application code providing the features seen by the user. From the guest's point of view, the framework represents a hardware abstraction layer that grants access to different subset of all devices in the IoT depending on the user's configuration. Using the concepts and conclusions from our analysis, the framework allows to use different virtualization strategies for the devices in the IoT system.

The configuration application, where the user can choose what hardware components to disable, runs as a second virtual machine, either on top of a small operating system or as a barebone application. Similar to the Dom0 virtual machine in the Xen hypervisor, it has special administrative privileges over the hypervisor, including the options to start and stop the guest and reconfigure the hypervisor. It is therefore part of the TCB. Alternatively, it would even be possible to merge the configuration app into the hypervisor; however, implementing the interface with all its required drivers might be easier in a small virtual machine. Together with the hypervisor, the configuration application and core constitute the trusted computing base of the IoT system, seen as just a HAL by the guest.

Similar to the BIOS setup menu, the configuration application is executed during the start of the device before the guest is booted. This makes sure that the guest cannot interfere with the configuration app itself, as there might be an overlap in required hardware. For example, the main IoT application and the configuration interface could use the same screen. In order to make the configuration app and the main IoT applications distinguishable, there must be at least one indicator, such as an LED, that is controlled by the TCB which signals to the user when the secure configuration interface is shown. This helps to prevent UI redress attacks where the user believes that they are interacting with the configuration interface, whereas in reality the interface shown to the user is imitated from within the guest VM. It is further important that the entire device is completely restarted before the guest is restarted with a new configuation in order to make sure that each controller accessible to the guest

is reset properly. If the IoT device is not restarted, a hardware component that is not reset by the guest at startup would continue to operate with the configuration written to it by the previous guest, which would violate our security requirements.

The configuration core is responsible for processing the user selection, determining what devices need to be accessible by the guest and finally reconfiguring the hypervisor. To this end, it contains a helper component for each virtualization driver, which takes care of configuring that virtualization driver for a given user selection and generating the corresponding device tree node for the guest.

In conclusion of our analysis in chapter 3, a subset of hardware components that may be disabled by the user must be defined during the development and implementation of this framework for a specific IoT device and appropriate virtualization drivers fulfilling all requirements mentioned in our analysis must be implemented.

4.2 Boot Sequence

Initially, the configuration application has full control over the required hardware using the passthrough assignment, therefore no device virtualization is required at this step. The configuration app is preconfigured to show a list of hardware components to the user, who can choose to disable them as required. Since the user sees the IoT device as a black box and does not know about its internal architecture, the interface should show meaningful descriptions of the features that a hardware provides. Internally, the configuration app maps the description shown to the user, for example `Camera left door`, to the corresponding node in the device tree, `/soc/csi@7e801000`. Dependencies between devices can be visualized either by nesting the dependent device into a submenu or, in more complex scenarios, by graying out those entries which cannot be enabled because of a disabled dependency.

After the user confirms the update system configuration of which devices are enabled, the configurator core runs our device assignment algorithm (see section 4.4) in order to

determine which hardware devices need to be disabled and therefore do not require any drivers to be loaded. Then, the hypervisor configuration must be updated to reflect the changes so we can be sure that only the required devices are mapped into the guest.

Before the guest is started, the configuration application also writes the selected hardware configuration, that is the list of enabled and disabled devices, to a permanent storage so that the configuration can persist over device reboots until the user decides to reconfigure the device. Lastly, the configuration changes must be communicated to the main operating system. Since the device tree is the de facto standard for device discovery in the ARM world, and because the configuration application and the configuration core internally use the device tree to represent all devices and emulation drivers, we decided to communicate the system's current device configuration to the guest via the device tree as well.

After that, the configuration application surrenders control of the hardware and is terminated so that the guest can be started. The hypervisor then loads and starts the guest virtual machine including the IoT operating system and application. Since at this point the configuration application is not executing anymore, we have avoided all device or resource conflicts between the configuration application and the guest. Further, besides central system components like the MMU, SMMU or GIC and paravirtualized systems, the hypervisor should not actively use any other device in order to minimize the need for shareable devices. For example, after loading the guest kernel into memory, and potentially a initial RAM filesystem, the hypervisor will not require access to any block storage or the boot medium anymore. Therefore, the block device drivers should be unloaded to avoid conflicts with the guest and otherwise unnecessary shareable virtualization drivers.

4.3 Configuration Sources

Our framework requires several configuration sources for different purposes, as depicted in figure 4.2. It also shows what configuration files are generated by the configuration core.

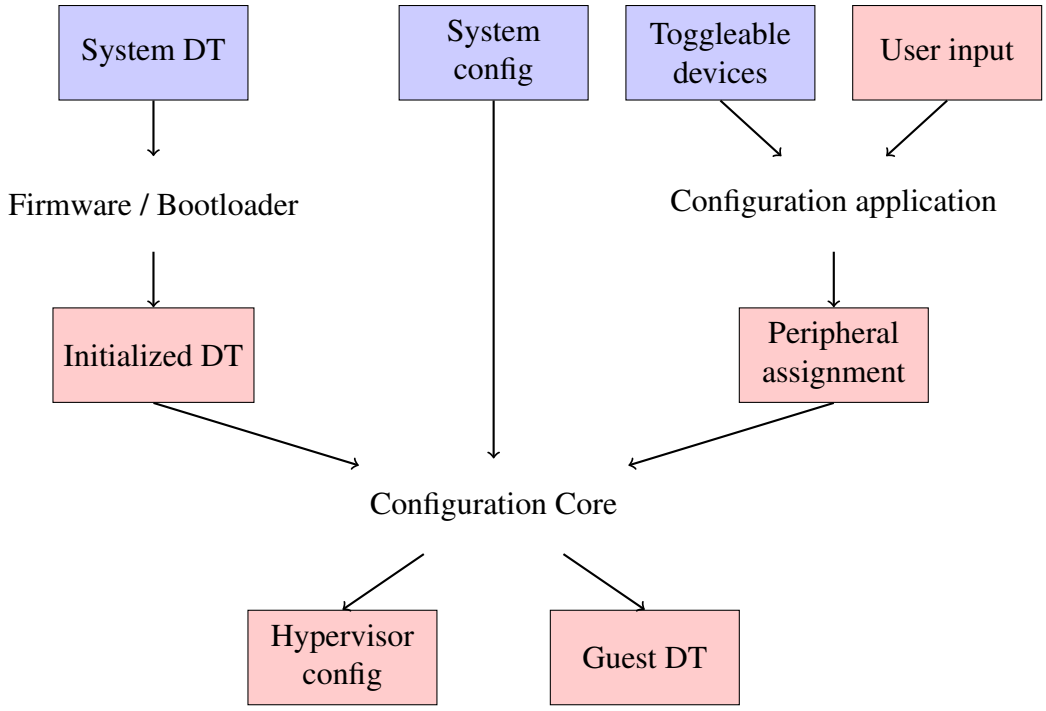


Figure 4.2: System configuration flow. The blue configurations are statically configured during the development of a system using the framework, while the red ones are dynamically determined at runtime.

4.3.1 System device tree

The System device tree is the device tree describing the hardware on the system. Since modifications may be applied to the system device tree by the firmware or bootloader during the boot, the configuration core must use the *initialized device tree* containing these changes.

4.3.2 System configuration

The system configuration consists of three configuration items. First, it contains an assignment of each device to either the TCB or the guest. Devices assigned to the guest can also be annotated as deactivatable, if the device is not required by the guest. All devices that the user may disable using the configuration application must be marked as deactivatable.

Second, the system configuration describes the virtualization type and driver to be used

for each device that can be exposed to the guest. This includes whether the virtualization strategy allows a device to be shared and whether it would be managed by the hypervisor or by the guest. Furthermore, similar to how each driver can be configured for each device in the device tree, options for each virtualization driver and device can be specified here.

The first two configuration options must also meet the requirements stated in our analysis with regard to accessibility of resources and services provided and consumed by devices in each domain.

Lastly, the system configuration also contains the base device tree for the guest, which contains some static configuration parameters. This can include the device tree's CPU nodes and other nodes not directly related to an actual hardware device such as the `/chosen` node specifying boot parameters.

4.3.3 Toggleable devices

The set of toggleable devices contains all devices which may be disabled by the user, which is only important for the configuration application. Combined with the user's choice, the configuration application will yield the *peripheral assignment* describing which hardware components should be functional and enabled for the guest and which should be disabled. This list must be a subset of all devices marked as deactivatable in the system configuration.

Devices such as an I^2C controller, which do not provide services directly to the user but connect peripherals providing features to the user, would be assigned to the guest and marked as deactivatable, but they would not be in the set of toggleable devices.

4.3.4 Generated configurations

After the configuration core runs the device assignment algorithm, it will generate two configuration files reflecting the user's choices:

Hypervisor configuration The hypervisor configuration defines which hardware resources and devices the guest can access. It also includes configuration options for all device

virtualization drivers.

Guest device tree The guest device tree is used by the guest's operating system for loading the corresponding drivers. Its generation from the base device tree contained in the system device tree is detailed in section 4.4.2.

4.4 Configuration Core

The heart of our framework is the configuration core, which assigns devices to the guest as required, regenerates the device trees for the guest and reconfigures the hypervisor.

4.4.1 Device Assignment

In order to minimize the number of enabled devices and the size of the TCB, the configuration core computes all hardware components reachable in the dependency graph starting from security critical devices, device always required by the guest and enabled peripherals. Following the design principle of least privilege [18], all devices not included in the set of reachable devices can be disabled, even if they have not been disabled explicitly by the user. For instance, using the Raspberry Pi 4 dependency graph as an example, if all devices on the I^2C bus are disabled, then the main I^2C controller `/soc/i2c@7e804000` can be disabled as well, since it is unused. Similarly, if there were any dependencies of this I^2C controller, which are not security critical and not used by any other enabled device, these could be disabled as well.

Since the system configuration must satisfy all dependencies and the rules of the virtualization strategy for each device accessible to the guest, the resulting assignment of devices into TCB, guest and disabled devices also satisfies all of these constraints.

4.4.2 Generating the Guest Device Tree

After each device has been assigned to the TCB, the guest or has been disabled, the device tree for the guest can be generated. To this end, the nodes corresponding to disabled devices

in the base guest device tree are simply removed, with one exception. As mentioned in the analysis, we can also disable devices which do not have any enabled consumers and are not marked as required by the TCB or the guest.

Then, the guest's device tree can be generated or translated node by node in a pre-order iteration of the device tree. In order to translate the node for the guest tree, we differentiate three cases based on whether or how the device corresponding to the device tree node is made accessible to the guest:

Passed-through or fully virtualized device : The contents of the device tree node are copied.

Paravirtualized device : The helper component of the virtualization driver generates a new device tree node based on the original node from the initialized device tree.

Not exposed to guest : The node is omitted from the guest device tree.

After translation, all references in properties of a node must be updated to point to the translated node. Then, the translated node is inserted into the guest device tree as a child of its translated parent.

However, when a parent-child relationship is not regarded as a dependency, this means that the parent node has no corresponding entry in the guest device tree. Since, in this case, the parent device has no significant function that is required to access the child, it is safe to replace it with a dummy `simple-bus` node that retains standardized device tree properties. The correctness of the resulting device tree follows from the fact that the dependency graph is a supergraph of the device tree with exception of some parent-child relationships which we handle separately.

4.4.3 Hypervisor Configuration

The helper components for each virtualization driver in the configuration core also each generate parts of the hypervisor configuration according to the user's selection. For example,

if the user decides to disable a device connected via I^2C, the corresponding address on the I^2C bus must be blacklisted by the I^2C virtualization driver.

Then, the hypervisor can configure the GIC, MMU and SMMU, if one is present, according to the requirements of each device that is enabled for the guest. To this end, for each device that is made accessible to the guest, the helper component of the used virtualization strategy defines what hardware resources must be made accessible to the guest. For guest managed devices, this includes the interrupts and MMIO regions specified for the device in the device tree. Hypervisor managed devices likely have different requirements depending on the virtualization strategy used. For example, the helper component of a paravirtualization driver can request different interrupts and a communication region in memory for the purpose of virtualizing a hardware component, instead of the original hardware resources specified for the virtualized device in the device tree.

4.5 Deactivating Devices Outside the Device Tree

In some cases, devices for which deactivation may be desired are not listed in the device tree, as we have shown with the example of the WiFi controller in the Raspberry Pi 4. Similarly, devices completely controlled in userspace usually do not have an entry in the device tree.

In order to support these devices for deactivation, pseudo entries without a `compatible` string may be added to the system device tree. Alternatively, in order to separate the actual system device tree from pseudo nodes, the hypervisor can load additional pseudo nodes as a device tree overlay before starting the configuration interface. Then, the helper component for the virtualization driver used for the bus in question must be able to infer based on these added nodes what resources or addresses to make available to the guest.

For example, in order to support deactivation of the WiFI controller in the Raspberry Pi 4, a pseudo node specifying the controllers address must be added to the device tree. Apart from that, an appropriate emulation driver for the MMC controller and the correct system configuration is required.

In the same fashion, PCI peripherals could be enabled and disabled as well, given required virtualization drivers.

CHAPTER 5

IMPLEMENTATION AND DISCUSSION

5.1 Implementation

5.1.1 Overview

We implemented the proposed framework for demonstration purposes using the Jailhouse hypervisor [19], a Linux hypervisor designed for static hardware partitioning in realtime systems using asynchronous multiprocessing (AMP). The configuration core is implemented in Python 3.

Jailhouse

Jailhouse combines the advantages of traditional Type-1 and Type-2 hypervisors into one system, see figure 5.1. First, the IoT system boots directly into a complete Linux system which has control over the entire hardware. We will call this system the host. This is similar to Type-2 hypervisors before the hypervisor is started. Then, it activates the hypervisor via a kernel module, which lifts itself below the operating system, taking control of the GIC and second level address translation. The main Linux system now runs inside a virtual machine called the root cell, which still has administrative control over the hypervisor similar to

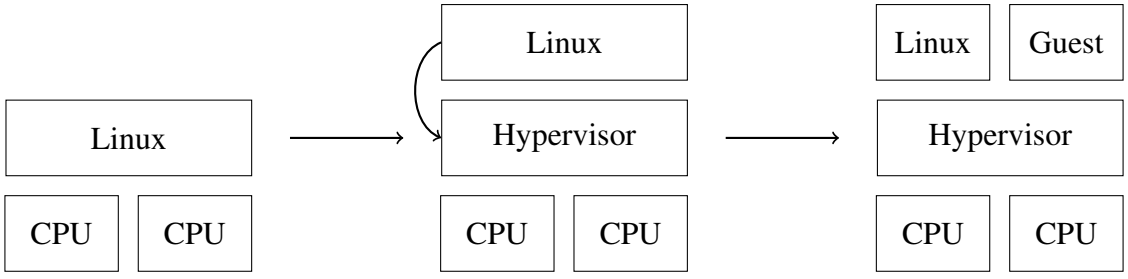


Figure 5.1: Jailhouse hypervisor initialization.

Dom0 in the Type-1 Xen hypervisor. As a guest virtual machine is started, called non-root cell in Jailhouse, the hardware resources assigned to it are taken away from the host. Jailhouse comes with support for SMMUs and the GIC. Selective PCI device passthrough by full emulation of a PCI Express (PCIe) controller is currently only supported on x86. It also supports passing through devices to guests and granting guests access to memory regions on a sub-page granularity. In addition, it already supports hardware device partition using pass-through assignment.

We chose the Jailhouse hypervisor because of its partitioning capabilities and the potential to reuse existing Linux device drivers for the host while also keeping the size of the hypervisor itself small. Further, the non-traditional design of Jailhouse allows us to use the main advantage of a Type-2 hypervisor for our purposes, namely being in full control of the entire hardware in order to show the configuration interface before the guest boots. Yet, at runtime of the guest the system is effectively powered by a Type-1 hypervisor. This is an effective method for sharing hardware between the configuration interface and the main IoT application without running into additional device or resource conflicts. This also means that the hypervisor itself requires no hardware drivers, except for the security critical devices such as the SMMU, MMU or GIC. As a result, drivers for devices such as memory cards or other block devices used as boot medium are not part of the hypervisor itself. Although this means that our implementation is constructed slightly differently compared to the proposed architecture in figure 4.1, the interface from the IoT operating system's point of view has not changed and in our opinion, the benefits of using the Jailhouse hypervisor in a practical implementation justify the internal changes in the TCB.

At runtime, our implementation therefore functions as shown in figure 5.2, as opposed to the architecture of the general framework displayed in figure 4.1. Most devices requiring emulation are controlled by the host, especially shared devices, since the host already supports a wide range of drivers. This significantly reduces the need for reimplementation of device drivers in the hypervisor. Device emulation is faciliated by the inter-VM com-

munication driver described in the following section. The hypervisor is only in control of system components like the MMU, SMMU and GIC. In addition, fully virtualized devices with all dependencies controlled by the hypervisor can also be virtualized by the hypervisor for better performance.

In order to prevent unnecessary device and resource conflicts, the host can be booted from an initial RAM filesystem containing device and virtualization drivers and the configuration application. The current configuration and the guest that will be booted are loaded from a storage device that is mounted by the operating system for this purpose and unmounted before the guest starts. Further, all devices and their drivers which are not required anymore are unloaded, rendering the entire host idle with the exception of the virtualization drivers.

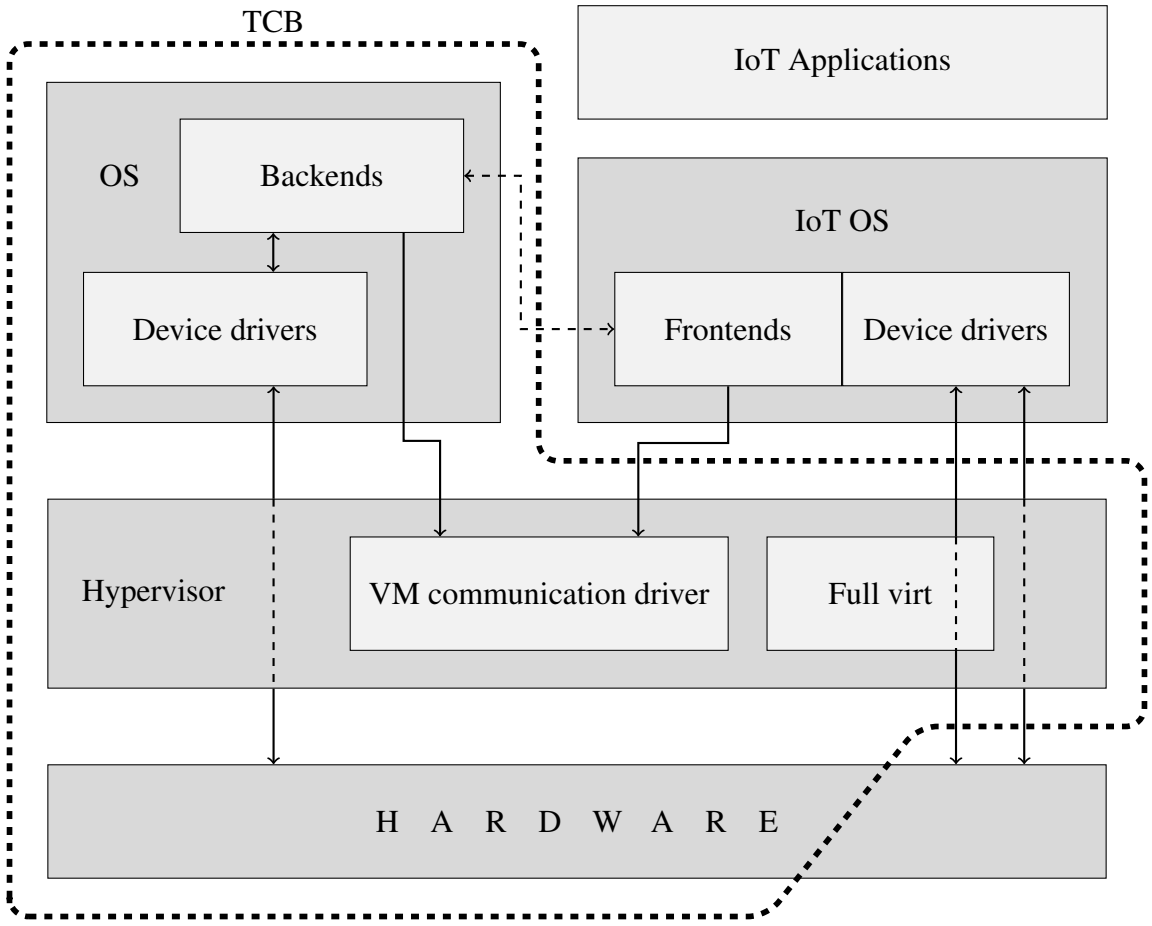


Figure 5.2: System architecture of our implementation after the guest has been started.

Inter-VM Communication Driver

Although Jailhouse already comes with an inter-VM shared memory (IVSHMEM) driver supporting emulation of ethernet devices or block devices, it requires an emulated PCIe device for this purpose and it is primarily designed to be used by applications. To support device emulation, we implemented a simpler communication driver in the hypervisor, which maps a configurable amount of memory into both the host and the guest. The driver allows each virtual machine to signal the other using an emulated doorbell register, which injects an interrupt in the other domain. Further, it can separate memory regions for multiple emulation drivers using the communication interface. The communication interface, including what memory regions are shared, are part of the general hypervisor configuration. What interrupt is used for signaling can be freely chosen by each virtual machine separately (guest and host) and changed at runtime within the interrupts enabled for a domain.

The interface of the kernel module, which can be used by emulation drivers, provides following functions:

- `init_device`: maps the memory regions associated with an emulated device identifier (ID). The memory region corresponding to the device ID must be configured in the hypervisor configuration.
- `exit_device`: unmaps the memory regions associated with an emulated device.
- `register_handler`: tells the driver to execute the passed function when it receives a notification for the emulated device from the other end of the emulation driver pair.
- `unregister_handler`: unregisters a notification handler.
- `notify`: notifies the driver on the other end of the emulation driver pair.

The memory mapped by the communication driver is synchronized and cached in the CPU's cache hierarchy. Any communication protocol can be used on top of the communi-

cation interface, for example I/O rings or the VirtIO transport protocol (without VirtIO's device layer since devices are statically described in the device tree) [20].

5.1.2 Configuration

The configuration structure is the same compared with the proposed framework, except that the configuration of the host system is an additional output of the configuration core. The main hypervisor configuration is stored as a template which can be adjusted based on the current configuration. The parts of the system configuration related to device assignment, emulation drivers and their (constant) configuration options are organized in a device tree overlay, which is merged with the host's device tree when the configuration core is initiated by the configuration application. We chose this design because of our similar requirements of per-device configuration and matching with emulation drivers. This overlay also specifies for each device, whether the parent-child relationship with its children must be regarded a dependency.

In order to load the correct virtualization drivers in the host, each helper component in the configuration core generates a device tree overlay based on a driver specific template. This node also contains the configuration options for the virtualization driver. After the user confirms their chosen device configuration, the drivers of devices that will be controlled by the guest are unloaded by the host and the devices removed from the host's device tree. Then, the overlays are applied to the host device tree before the guest is started, so that all emulated devices are ready to be used. By using this approach, we can also load the same virtualization driver for multiple devices of the same kind.

5.1.3 Extracting Dependencies from the Device Tree

Since the initialized device tree, from which we want to extract device dependencies, comes in its compiled form, higher level concepts in properties, such as strings or references, cannot be reconstructed for custom properties, as the interpretation is up to the compatible

drivers. In order to support extracting all dependencies even for device specific properties, we modified the device tree compiler to insert a special `__dependencies__` property into each node in the device tree. Similar to how debug information can be added to binary programs during compilation, this property lists the location of each reference as a byte offset within the property that contains the reference. This approach allows reconstruction of all node dependencies from a compiled device tree at runtime, even the ones that have been updated by the firmware or bootloader.

5.2 Example: Raspberry Pi 4

To give a demonstration of the framework and our implementation, we implemented virtualization strategies for two hardware components of the Raspberry Pi 4. We also attached a Sense HAT board to simulate an external integrated circuit connected via an I^2C interface.

One hardship was the fact that the firmware executing before the bootloader expects all hardware components of the standard configuration to be enabled. In order to prevent the host to load built-in drivers for disabled devices, we added a script to the bootloader which marks each node that corresponds to a deactivated device as disabled. In the same way we experienced boot problems when the `__dependencies__` property was added to each node. As a workaround, these properties are loaded as an overlay just before the configuration core is started.

5.2.1 Example Fully Virtualized Device: Ethernet Controller

Our first hardware component which we made available to the guest is the Raspberry Pi's ethernet controller, `/scb/genet@7d5800000`. Although the user could simply unplug the ethernet cable in order to prevent communication via the ethernet connection, we chose this example as a proof for how a fully virtualized device can be made accessible to the guest. The dependency graph of the Raspberry Pi 4 (figure A.2) reveals that the hardware component only depends on the system's interrupt controller and a grandchild node. Since

all its descendants do not have any other external dependencies, the three nodes can be seen as one functional unit.

As a matter of fact, out of our list of candidate devices for deactivation, this is the only hardware component in the Raspberry Pi 4 that only depends on the GIC and therefore can be virtualized using a guest managed technique without requiring virtualization of another device.

Because the ethernet controller contains integrated DMA capabilities, we must use full virtualization in order to prevent DMA attacks. To this end, we implemented a generic DMA filer mechanism in Jailhouse based on the trap and emulate principle. It intercepts all accesses to emulated MMIO registers which accept parts of an DMA descriptor (address and size of a device-to-memory or memory-to-device DMA transaction). At the same time, this filter also translates the DMA address from guest physical address space into the machine physical address space, thereby simulating the functionality of an SMMU. All descriptors that reference regions not mapped to the VM are discarded as an attempted DMA attack.

In case of the Raspberry Pi's ethernet controller, DMA descriptors can be written to two MMIO regions at offsets `0x2000` and `0x4000` from the first register of the device, so `0xfd582000` and `0xfd584000`. Each region contains 768 registers for 256 consecutive DMA descriptors consisting of a 32 bit size register and two registers for the lower and upper 32 bits of a 64 bit address.

The child MDIO bus and the `genet-phy` controller on it are neither memory mapped nor able to initiate DMA transactions, both of the ethernet controller's descendants can therefore be passed through. Since all three components are managed by the guest, all of their dependencies are satisfied.

In the end, the ethernet controller is accessible from the guest and the guest is able to connect to a local network with Gigabit link speed. We noticed that the ethernet controller cannot be used when the guest is restarted without restarting the entire system including the host, which could indicate that the device driver does not shut down the controller

```
genet@7d580000 {
  jhmgr,strategy = <(STRATEGY_EMULATE
                     | STRATEGY_GUEST_MANAGED)>;
  jhmgr,compatible = "dma-proxy";
  jhmgr,dmadescs =
      <0x2000 0x4 0x0 0xc 256 0xc
       (JAILHOUSE_DMADESC_ADDR_SPLIT
       | JAILHOUSE_DMADESC_TRIGGER_ADDR_H
       | JAILHOUSE_DMADESC_TRIGGER_SIZE)>,
      <0x4000 0x4 0x0 0xc 256 0xc ..>;
  __dependencies__ = "phy-handle", <0>;

  mdio@e14 {
    jhmgr,strategy = <STRATEGY_PASSTHROUGH>;

    genet-phy@0 {
      jhmgr,strategy = <STRATEGY_PASSTHROUGH>;
    };
  };
};
```

Figure 5.3: Configuration for the Raspberry Pi 4 ethernet controller in the system configuration.

properly. Also, after powering down the guest virtual machine only and restarting it with the ethernet controller disabled, the controller still seems to process incoming network packets as indicated by the blinking activity LED next to the socket. However, once the entire device is restarted including the host, the controller is inactive. Since the it is not part of any power domain that could be powered off, the ethernet controller is still powered at all times.

5.2.2 Example Virtualized Device: I2C Bus Controller

The main I^2C controller that is active by default, `/soc/i2c@7e804000g`, is connected to the Sense HAT extension in our setup. The Sense HAT board simulates an external circuit board with 5 connected I^2C devices, 4 sensors and the pixel grid including joystick, all of which can be controlled separately. To allow selectively disabling these devices, the I^2C controller must be emulated.

Each address on the I^2C bus can be thought of as a separate resource provided by the device. The I^2C controller also supports slave mode, which allows connected I^2C devices to directly communicate with each other. Therefore, the use of this mode by the guest must be prevented.

Since the I^2C controller depends on the main hardware clock and the GPIO controller, it must be paravirtualized from the host. This also means we can reuse the main I^2C driver and emulate the device at the interface level of the I^2C subsystem in Linux. Devices accessed via the I^2C bus can be, dependency permitting, passed through.

To make the Sense HAT controllable from the guest, we had to apply a small patch for a known issue with the device drivers[1]. In addition, we patched the driver to be modular so that the pixel grid can be used independently of the joystick, making the dependency on the GPIO controller optional.

In the end, we are able to separately enable or disable the pressure sensor, humidity sensor, magnetic sensor, acceleration sensor on the Sense HAT individually. The pixel matrix can only be enabled or disabled together with the joystick, since they share are controlled using the same I^2C bus address.

5.2.3 Other Raspberry Pi Components

Similar to how we granted the guest access to the ethernet controller and the I^2C controller, this could also be done for the PCIe controller and SPI controllers. Jailhouse natively supports PCI emulation, which can be used to grant the guest access to a subset of the connected PCI devices. However, at the time of writing, Jailhouse's support for PCI passthrough on ARM was not fully functional yet. Further, the enabled PCI devices would require emulation or mediation by an SMMU in the real world, since they are able to access the entire physical address space with DMA by using the PCIe controller as a bridge.

As with I^2C, SPI also allows multiple devices connected to the same SPI controller. For

[1] `https://github.com/raspberrypi/linux/issues/3300`

this reason, it is desirable to fully emulate the SPI controller in the same way as the I^2C controller to facilitate separate disabling of connected circuits.

The other peripheral controllers in the Raspberry Pi 4, all of which are in the top section of the device graph in figure A.2, also fulfill our requirements for devices that should be deactivatable defined in our analysis. In order to deactivate those devices, the central hardware components `/soc/gpio@7e200000` and `/soc/cprman@7e101000` would need to be paravirtualized, since we already determined in section 3.7 that paravirtualizing the deactivatable devices themselves does not yield the potential reduction in attack surface that we intend to get. In addition, the DMA controller `/soc/dma@7e007000` must be made available to the guest using full or paravirtualization. Then, each of the peripheral controllers can either be passed through or fully virtualized to filter out rogue DMA transactions.

5.3 Evaluation

For evaluation of our example implementation of the framework, we will compare the implementation with each of the security requirements initially stated in the introduction.

5.3.1 Tamper Proof

The configuration application and the guest are temporally isolated, and the configuration itself cannot be accessed as they are stored on the SD card which is not accessible to the guest. All boot programs and their configuration is stored on the SD card as well. As a result, the guest cannot tamper with the configuration of the TCB.

5.3.2 Complete Mediation

We established in the background and analysis chapters what security primitives are available to us on the ARM platform and to what extent they can enforce the boundary between the TCB and the operating system. The MMC controllers used for accessing the SD card that stores the system configuration is part of the TCB and all facts established in the following

apply to it as a result.

The memory that belongs to the TCB, comprised of the hypervisor's and the host's memory, is completely shielded from regular memory accesses by the guest VM, since the guest operates in a virtual address space that does not map the TCB's memory regions. All devices that we make available to the guest either cannot initiate DMA transactions, which includes the devices on the I^2C bus, or are fully or paravirtualized in order to prevent and filter out DMA attacks. Similarly, only the interrupts associated with passed through devices and fully virtualized devices are accessible to the guest VM, which is enforced by the configuration of the vGIC through the hypervisor. Furthermore, even when all of the Raspberry Pi's peripheral controllers are enabled for the guest using the method described above, paravirtualization of the central hardware components ensures that the guest cannot interfere with the resources provided by these devices to those devices controlled by the TCB.

It follows from all these facts that the guest cannot tamper with any hardware resource consumed by the TCB, as well as the TCB itself or its configuration.

5.3.3 Correctness

Determining the correctness of our system is challenging and almost impossible to do with perfect accuracy. Instead, we will use the size and complexity of the TCB in terms of lines of code, and especially the complexity of its interfaces with the guest as a proxy for its correctness. Table 5.1 breaks down the size of our implementation into different software components[2].

Each component listed in the table is of reasonable size. Although the Jailhouse hypervisor contains a lot of code, it contains mostly drivers for core hardware components such as the GIC, MMU and SMMU, as well as the unused PCIe driver and virtualization driver. Further, our communication driver contains barely any complexity, as it is only glue code

[2]Lines of code were determined using the tool `sloccount`

Table 5.1: Lines of code for each component of our implementation.

Component	**Language**	**LOC**
Jailhouse (arm64)	C, Assembly	5,423
VM communication driver	C	671
- Frontend	C	321
- Backend	C	350
I^2C device driver (Linux)	C	478
I^2C paravirtualization driver	C	966
- Frontend	C	460
- Backend	C	506
DMA filter + translation	C	304
Configuration core	Python 3	1,469
Configuration app	Python 3	112
Total	C, Assembly, Python 3	9,423
- TCB	C, Assembly, Python 3	8,642

for mapping statically configured memory regions into the VM's address space, injecting interrupts, and provides the small interface for kernel drivers described in section 5.1.1. Due to the temporal isolation of the configuation application and the guest, the configuration application and core cannot be interacted with by the assumed remote attacker, even when the guest is compromised. Further, because of the very limited input options by the user, all error states in the configuration application and core should have been covered by our extensive testing.

The biggest issue with the implementation may be the fact that an entire Linux system is running in the TCB throughout the lifetime of the system, and the guest is even able to interact with it via the virtualization drivers. However, we argue that the actual interface that can be used by the guest, the I^2C paravirtualization driver, is sufficiently small to prevent exploitation. About half of the I^2C virtualization driver consists of device setup and teardown code as well as handlers for the simple I^2C interface functions, including bus locking, unlocking, transferring a SMBus message and reading the device capabilities. The most complex part is the serializing and deserializing code of 469 lines.

In conclusion, we have reason to believe that our implementation is correct.

5.3.4 Availability / No Interference

Because of the temporal isolation of the configuration application and the guest, the configuration application is guaranteed to operate correctly.

5.4 Discussion

The implementation and evaluation shows that running an entire Linux kernel in the host indeed has merits and drawbacks. While the effect is rather small in our example due to the simplicity of the I^2C driver, being able to reuse device drivers that are already implemented in Linux can be a great advantage for complex devices that must be supported by the TCB, regardles of whether the hardware component is required by the configuration application or must be paravirtualized for the guest. This greatly reduces the chance of bugs being introduced by reimplementing a device driver. Further, the paravirtualization driver must only be implemented once for each subsystem of the kernel and can be reused thereafter, meaning that a larger community can maintain the correctness of its implementation as opposed to just a single vendor or developer in case of a device-specific full virtualization driver. One way to reduce the vast complexity of the LInux kernel would be to use a hardened kernel.

Further, we also reduced the complexity of the paravirtualization transport as compared to common solutions like VirtIO [20]. While VirtIO, just like Jailhouse's builtin IVSHMEM driver, uses emulated buses such as PCI to allow easy dynamic configuration via bus probing, our approach does not require the additional emulation of an entire generic PCI controller, which is even more beneficial when the hardware does not provide a PCI controller that would warrant PCI emulation to restrict the accessible PCI devices.

5.4.1 Limitations

At the same time, there are a small number of limitations to our current implementation.

First, it makes extensive use of the ARM Virtualization Extensions, which may not be available on all systems where an implementation of our framework would be desired. For example, ARM Cortex-M based processors do not provide any hardware virtualization support. While we have not tested it, a trap and emulate or system-wide paravirtualization approach should work in place of the hardware virtualization support, however with significant performance overhead.

Second, with our implementation most paravirtualization would probably be implemented using the same interface provided by the HAL of the Linux kernel for a specific device class, just like we did for the I^2C controller. This means that the guest is limited to running the same version of Linux in order to make sure that it is compatible with the interface exposed via paravirtualization. However, any other operating system, including barebone applications, could be run in the guest virtual machine with the required frontend driver implementation.

Third, with the current assignment algorithm, the I^2C controller is paravirtualized even when factory settings are used where all devices on the I^2C bus are enabled. In this case, paravirtualization only adds overhead and increases the complexity of the TCB without adding any benefits until a device on the I^2C bus is disabled.

Fourth, all deactivated devices in our example are not actually powered down and continue to operate, just without any interaction with the IoT operating system. For peripheral devices that do not provide any communication channel to the outside world, this means the device is consuming power while not being usable. However, for controllers facing the outside world like the ethernet controller, the controller firmware is still being executed by the controller and as a result, this setup would not guarantee that attackers cannot exploit vulnerabilities in the controller's firmware.

Lastly, the correctness of our dependency analysis, and thereby our entire assignment algorithm and overall operability of the hardware by both the TCB and the guest, depends on the assumption that we can predict what devices are present in an IoT system and what

their dependencies are. However, boot programs executed before the hypervisor, or Linux in our implementation, may arbitrarily change the device tree, including adding nodes or introducing dependencies between devices. Unless the boot program cannot add any nodes that are not contained in the system configuration, and entries for each added reference are added to each node's `__dependencies__` property, added devices are hidden from the guest and it is possible that some devices do not function properly because the dependency analysis yields incomplete or wrong results. For devices like the Raspberry Pi which have a closed source firmware, this guarantee cannot be made.

CHAPTER 6
RELATED WORK

Apart from our framework, we briefly present other research on related problems.

6.1 Notary

Notary is a security-by-design software architecture for systems with very strict security requirements, such as hardware wallets for cryptocurrencies or transaction approval agents [21]. Notary achieves strong isolation and compartmentalization of system components by executing them on physically separate CPUs. Software running on the system is further simplified by resetting the entire CPU and using a deterministic startup sequence for each program that allows the verification of the program's correctness and security. Thereby, Notary overcomes classes of bugs like vulnerabilities in the OS and side channels.

6.2 LTZvisor

LTZvisor is an experimental hypervisor that uses ARM TrustZone as an hardware backed isolation mechanism as opposed to ARM Virtualization extensions [22, 23]. The hypervisor supports both versions of ARM TrustZone, the more powerful variant on ARM Cortex-A and the more simplistic one on ARM Cortex-M processors, and thereby can be used on a wide range of IoT boards. At the moment, however, only a few development boards are supported.

Further research has been conducted by the same authors towards enabling efficient communication between software partitioned into the two worlds [24].

CHAPTER 7
CONCLUSION

In this thesis, we presented a framework for deactivating single hardware components by choice of the user, either in order to proactively reduce the overall attack surface of the system or to retroactively mitigate known vulnerabilities. We formalized how hardware components in complex ARM systems might interfere with each other and how this is manifested in the device tree using the example of the Raspberry Pi 4. Using our insights, we developed an extraction and visualization tool for device dependencies to ease correct implementation of our framework and showed how virtualization is key to enforcing the hardware partition between activated and deactivated devices. We further analyed and defined requirements and rules for implementing our selective deactivation framework by combining I/O device passthrough, full virtulalization and paravirtualization while ensuring proper operability and security of the entire system with regard to our initial goals. Lastly, we gave a proof of concept implementation that allows selectively disabling 6 different components on the Raspberry Pi 4 and how similar effort can be used for other peripherals.

7.1 Future Work

There are several ways how the presented framework can further be extended. As we have already mentioned in our discussion, the current implementation is limited to systems with ARM Virtualization Extensions. By also supporting ARM TrustZone enabled virtualiation as shown by Pinto et al., a larger range of devices could be supported. Integrating TrustZone capabilities would also allow adding another hardware backed method for enabling I/O devices which can initiate DMA transactions to the guest without adding any device specific code.

The framework can also be extended by a TCB backed factory reset mechanism for the

untrusted guest in case of a suspected infection of the IoT application. Further integration with the power management controllers and reset mechanisms could be used to power down unused devices and reduce the energy consumption of the IoT system. This would be especially useful for deactivating controllers interacting with remote entities such as Bluetooth, WiFi or ethernet controllers, since the controller's firmware might still interact with adversaries.

Lastly, a more complex assignment algorithm, which can optimize the virtualization strategy used for each hardware components by choosing from a pool of strategies for each component, could greatly reduce the added overhead to the system when factory settings are used or most devices are enabled.

Appendices

APPENDIX A

DEVICE TREE AND DEPENDENCIES

A.1 Raspberry Pi 4 Device Tree Source Code

```
/dts-v1/;
/memreserve/ 0 0x1000;
/ {
  compatible = "raspberrypi,4-model-b", "brcm,bcm2711";
  model = "Raspberry Pi 4 Model B";
  interrupt-parent = <&gicv2>;
  #address-cells = <2>;
  #size-cells = <1>;

  chosen {
    bootargs = "coherent_pool=1M 8250.nr_uarts=1 cma=64M";
  };

  soc {
    compatible = "simple-bus";
    #address-cells = <1>;
    #size-cells = <1>;
    ranges = <0x7e000000 0x0 0xfe000000 0x01800000>,
             <0x7c000000 0x0 0xfc000000 0x02000000>,
             <0x40000000 0x0 0xff800000 0x00800000>;
    dma-ranges = <0xc0000000 0x0 0x0 0x3c000000>;

    dma: dma@7e007000 {
      compatible = "brcm,bcm2835-dma";
      reg = <0x7e007000 0xb00>;
      interrupts = <GIC_SPI 80 IRQ_TYPE_LEVEL_HIGH>, ..;
      interrupt-names = "dma0", "dma1", ..;
      #dma-cells = <1>;
      brcm,dma-channel-mask = <0x1f5>;
    };

    pm: watchdog: watchdog@7e100000 {
      compatible = "brcm,bcm2835-pm", "brcm,bcm2835-pm-wdt";
      #power-domain-cells = <1>;
```

```
35        #reset-cells = <1>;
36        reg = <0x7e100000 0x114>,
37              <0x7e00a000 0x24>,
38              <0x7ec11000 0x20>;
39        clocks = <&clocks BCM2835_CLOCK_V3D>, ..;
40        clock-names = "v3d", ..;
41        system-power-controller;
42      };
43
44      clocks: cprman@7e101000 {
45        compatible = "brcm,bcm2711-cprman";
46        #clock-cells = <1>;
47        reg = <0x7e101000 0x2000>;
48        clocks = <&clk_osc>, <&dsi0 0>, <&dsi1 0>, ..;
49        firmware = <&firmware>;
50      };
51
52      mailbox: mailbox@7e00b880 {
53        compatible = "brcm,bcm2835-mbox";
54        reg = <0x7e00b880 0x40>;
55        interrupts = <GIC_SPI 33 IRQ_TYPE_LEVEL_HIGH>;
56        #mbox-cells = <0>;
57      };
58
59      gpio: gpio@7e200000 {
60        compatible = "brcm,bcm2711-gpio", "brcm,bcm2835-gpio";
61        reg = <0x7e200000 0xb4>;
62        interrupts = <GIC_SPI 113 IRQ_TYPE_LEVEL_HIGH>, ..;
63        gpio-controller;
64        #gpio-cells = <2>;
65        interrupt-controller;
66        #interrupt-cells = <2>;
67        pinctrl-names = "default";
68
69        emmc_gpio48: emmc_gpio48 {
70          brcm,pins = <48 49 50 51 52 51>;
71          brcm,function = <BCM2835_FSEL_ALT3>;
72        };
73
74        sdhost_gpio48: sdhost_gpio48 {
75          brcm,pins = <22 23 24 25 26 27>;
76          brcm,function = <BCM2835_FSEL_ALT0>;
77        };
78
79        i2s_pins: i2s {
```

```
80          brcm,pins = <18 19 20 21>;
81          brcm,function = <BCM2835_FSEL_ALT0>;
82        };
83
84        i2c1_pins: i2c1 {
85          brcm,pins = <2 3>;
86          brcm,function = <BCM2835_FSEL_ALT0>;
87          brcm,pull = <BCM2835_PUD_UP>;
88        };
89
90        sdio_pins: sdio_pins {
91          brcm,pins = <34 35 36 37 38 39>;
92          brcm,function = <BCM2835_FSEL_ALT3>;
93          brcm,pull = <0 2 2 2 2 2>;
94        };
95
96        uart1_pins: uart1_pins {
97          brcm,pins;
98          brcm,function;
99          brcm,pull;
100       };
101     };
102
103     i2s: i2s@7e203000 {
104       compatible = "brcm,bcm2835-i2s";
105       reg = <0x7e203000 0x24>;
106       clocks = <&clocks BCM2835_CLOCK_PCM>;
107       dmas = <&dma 2>, <&dma 3>;
108       dma-names = "tx", "rx";
109       status = "disabled";
110       #sound-dai-cells = <0>;
111       pinctrl-names = "default";
112       pinctrl-0 = <&i2s_pins>;
113     };
114
115     i2c1: i2c@7e804000 {
116       compatible = "brcm,bcm2835-i2c";
117       reg = <0x7e804000 0x1000>;
118       interrupts = <GIC_SPI 117 IRQ_TYPE_LEVEL_HIGH>;
119       clocks = <&clocks BCM2835_CLOCK_VPU>;
120       #address-cells = <1>;
121       #size-cells = <0>;
122       status = "disabled";
123       pinctrl-names = "default";
124       pinctrl-0 = <&i2c1_pins>;
```

```
125         clock-frequency = <100000>;
126
127         rpi-sense@46 {
128           compatible = "rpi,rpi-sense";
129           reg = <0x46>;
130           keys-int-gpios = <&gpio 23 1>;
131           status = "okay";
132         };
133
134         lsm9ds1-magn@1c {
135           compatible = "st,lsm9ds1-magn";
136           reg = <0x1c>;
137           status = "okay";
138         };
139
140         lsm9ds1-accel6a {
141           compatible = "st,lsm9ds1-accel";
142           reg = <0x6a>;
143           status = "okay";
144         };
145
146         lps25h-press@5c {
147           compatible = "st,lps25h-press";
148           reg = <0x5c>;
149           status = "okay";
150         };
151
152         hts221-humid@5f {
153           compatible = "st,hts221-humid", "st,hts221";
154           reg = <0x5f>;
155           status = "okay";
156         };
157       };
158
159       csi1: csi@7e801000 {
160         compatible = "brcm,bcm2835-unicam";
161         reg = <0x7e801000 0x800>,
162               <0x7e802004 0x4>;
163         interrupts = <GIC_SPI 103 IRQ_TYPE_LEVEL_HIGH>;
164         clocks = <&clocks BCM2835_CLOCK_CAM1>;
165         clock-names = "lp";
166         #address-cells = <1>;
167         #size-cells = <0>;
168         #clock-cells = <1>;
169         status = "disabled";
```

```
170         power-domains = <&power RPI_POWER_DOMAIN_UNICAM1>;
171
172         port {
173           endpoint {
174             data-lanes = <1 2>;
175           };
176         };
177       };
178
179       dsi0: dsi@7e209000 {
180         compatible = "brcm,bcm2835-dsi0";
181         reg = <0x7e209000 0x78>;
182         interrupts = <GIC_SPI 100 IRQ_TYPE_LEVEL_HIGH>;
183         #address-cells = <1>;
184         #size-cells = <0>;
185         #clock-cells = <1>;
186         clocks = <&clocks BCM2835_PLLA_DSI0>, ..;
187         clock-names = "phy", "escape", "pixel";
188         clock-output-names = "dsi0_byte", ..;
189         power-domains = <&power RPI_POWER_DOMAIN_DSI0>;
190       };
191
192       dsi1: dsi@7e700000 {
193         compatible = "brcm,bcm2835-dsi1";
194         reg = <0x7e700000 0x8c>;
195         interrupts = <GIC_SPI 108 IRQ_TYPE_LEVEL_HIGH>;
196         #address-cells = <1>;
197         #size-cells = <0>;
198         #clock-cells = <1>;
199         clocks = <&clocks BCM2835_PLLD_DSI1>, ..;
200         clock-names = "phy", "escape", "pixel";
201         clock-output-names = "dsi1_byte", ..;
202         status = "disabled";
203         power-domains = <&power RPI_POWER_DOMAIN_DSI1>;
204       };
205
206       dpi@7e208000 {
207         compatible = "brcm,bcm2835-dpi";
208         reg = <0x7e208000 0x8c>;
209         clocks = <&clocks BCM2835_CLOCK_VPU>, ..;
210         clock-names = "core", "pixel";
211         #address-cells = <1>;
212         #size-cells = <0>;
213         status = "disabled";
214       };
```

```
215
216     aux: aux@7e215000 {
217       compatible = "brcm,bcm2835-aux";
218       #clock-cells = <1>;
219       reg = <0x7e215000 0x8>;
220       clocks = <&clocks BCM2835_CLOCK_VPU>;
221     };
222
223     uart1: serial@7e215040 {
224       compatible = "brcm,bcm2835-aux-uart";
225       reg = <0x7e215040 0x40>;
226       interrupts = <GIC_SPI 93 IRQ_TYPE_LEVEL_HIGH>;
227       clocks = <&aux BCM2835_AUX_CLOCK_UART>;
228       status = "okay";
229       pinctrl-names = "default";
230       pinctrl-0 = <&uart1_pins>;
231     };
232
233     spi0: spi@7e204000 {
234       compatible = "brcm,bcm2835-spi";
235       reg = <0x7e204000 0x200>;
236       interrupts = <GIC_SPI 118 IRQ_TYPE_LEVEL_HIGH>;
237       clocks = <&clocks BCM2835_CLOCK_VPU>;
238       dmas = <&dma 6 &dma 7>;
239       dma-names = "tx", "rx";
240       #address-cells = <1>;
241       #size-cells = <0>;
242       status = "disabled";
243       pinctrl-names = "default";
244       pinctrl-0 = <&spi0_pins &spi0_cs_pins>;
245       cs-gpios = <&gpio 8 1>, <&gpio 7 1>;
246
247       spidev@0 {
248         compatible = "spidev";
249         reg = <0x0>;
250         #address-cells = <1>;
251         #size-cells = <0>;
252         spi-max-frequency = <125000000>;
253       };
254
255       spidev@1 {
256         compatible = "spidev";
257         reg = <0x1>;
258         #address-cells = <1>;
259         #size-cells = <0>;
```

```
260         spi-max-frequency = <125000000>;
261       };
262     };
263
264     gicv2: gic400@40041000 {
265       interrupt-controller;
266       #interrupt-cells = <3>;
267       compatible = "arm,gic-400";
268       reg = <0x40041000 0x1000>,
269             <0x40042000 0x2000>,
270             <0x40044000 0x2000>,
271             <0x40046000 0x2000>;
272       interrupts = <GIC_PPI 0 ..>;
273     };
274
275     emmc2: emmc2@7e340000 {
276       compatible = "brcm,bcm2711-emmc2";
277       status = "okay";
278       interrupts = <GIC_SPI 126 IRQ_TYPE_LEVEL_HIGH>;
279       clocks = <&clocks BCM2711_CLOCK_EMMC2>;
280       reg = <0x7e340000 0x100>;
281       broken-cd;
282       vqmmc-supply = <&sd_io_1v8_reg>;
283       vmmc-supply = <&sd_vcc_reg>;
284     };
285
286     mmcnr: mmcnr@7e300000 {
287       compatible = "brcm,bcm2835-mmc", "brcm,bcm2835-sdhci";
288       reg = <0x7e300000 0x100>;
289       interrupts = <GIC_SPI 126 IRQ_TYPE_LEVEL_HIGH>;
290       clocks = <&clocks BCM2835_CLOCK_EMMC>;
291       dmas = <&dma 11>;
292       dma-names = "rx-tx";
293       brcm,overclock-50 = <0>;
294       non-removable;
295       status = "okay";
296       pinctrl-names = "default";
297       pinctrl-0 = <&sdio_pins>;
298       bus-width = <4>;
299     };
300
301     firmware: firmware {
302       compatible = "raspberrypi,bcm2835-firmware", ..;
303       mboxes = <&mailbox>;
304
```

```
305       expgpio: gpio {
306         compatible = "raspberrypi,firmware-gpio";
307         gpio-controller;
308         #gpio-cells = <2>;
309         gpio-line-names = "BT_ON", "WL_ON", ..;
310         status = "okay";
311       };
312     };
313
314     power: power {
315       compatible = "raspberrypi,bcm2835-power";
316       firmware = <&firmware>;
317       #power-domain-cells = <1>;
318     };
319
320     [..]
321   };
322
323   leds {
324     compatible = "gpio-leds";
325
326     act_led: act {
327       label = "led0";
328       default-state = "keep";
329       linux,default-trigger = "mmc0";
330       gpios = <&gpio 42 GPIO_ACTIVE_HIGH>;
331     };
332
333     pwr_led: pwr {
334       label = "led1";
335       linux,default-trigger = "default-on";
336       gpios = <&expgpio 2 GPIO_ACTIVE_LOW>;
337     };
338   };
339
340   clocks {
341     compatible = "simple-bus";
342     #address-cells = <1>;
343     #size-cells = <0>;
344
345     clk_osc: clock@3 {
346       compatible = "fixed-clock";
347       reg = <0x3>;
348       #clock-cells = <0>;
349       clock-output-names = "osc";
```

```
350       clock-frequency = <54000000>;
351     };
352   };
353
354   arm-pmu {
355     compatible = "arm,cortex-a72-pmu", "arm,cortex-a15-pmu";
356     interrupts = <GIC_SPI 16 IRQ_TYPE_LEVEL_HIGH>, ..;
357     interrupt-affinity = <&cpu0>, <&cpu1>, <&cpu2>, <&cpu3>;
358   };
359
360   timer {
361     compatible = "arm,armv7-timer";
362     interrupts = <GIC_PPI 13 ..>, ..;
363     arm,cpu-registers-not-fw-configured;
364     always-on;
365   };
366
367   cpus {
368     #address-cells = <1>;
369     #size-cells = <0>;
370     enable-method = "brcm,bcm2836-smp";
371
372     cpu0: cpu@0 {
373       device_type = "cpu";
374       compatible = "arm,cortex-a72";
375       reg = <0x0>;
376       enable-method = "spin-table";
377       cpu-release-addr = <0x0 0x000000d8>;
378     };
379
380     [..]
381   };
382
383   scb: scb {
384     compatible = "simple-bus";
385     #address-cells = <2>;
386     #size-cells = <1>;
387     ranges = <0x0 0x7c000000 0x0 0xfc000000 0x03800000>,
388              <0x0 0x40000000 0x0 0xff800000 0x00800000>,
389              <0x6 0x00000000 0x6 0x00000000 0x40000000>,
390              <0x0 0x00000000 0x0 0x00000000 0xfc000000>;
391     dma-ranges = <0x0 0x0 0x0 0x0 0xfc000000>;
392
393     pcie_0: pcie@7d500000 {
394       reg = <0x0 0x7d500000 0x9310>,
```

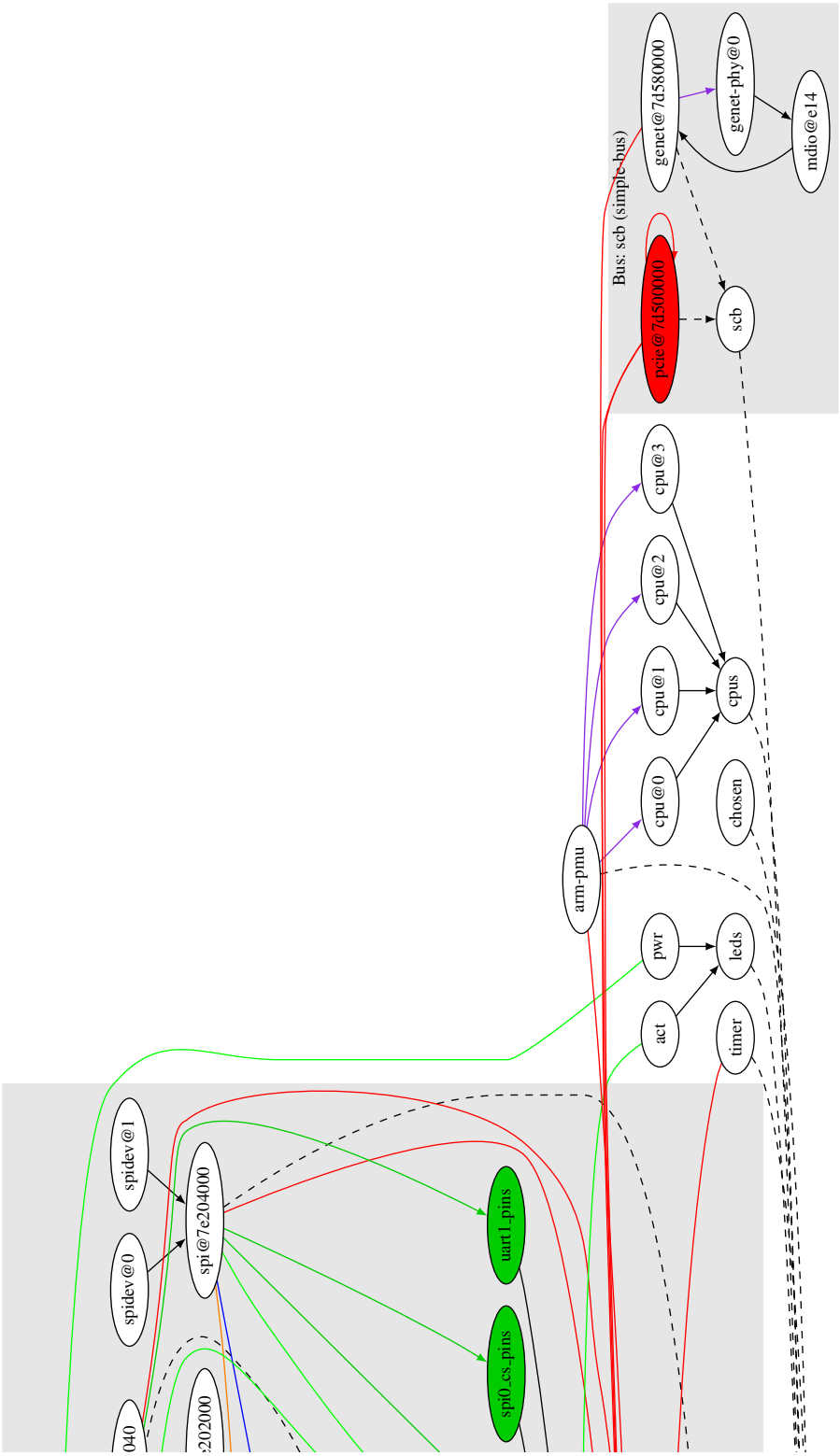
```
395             <0x0 0x7e00f300 0x20>;
396       msi-controller;
397       msi-parent = <&pcie_0>;
398       #address-cells = <3>;
399       #interrupt-cells = <1>;
400       #size-cells = <2>;
401       bus-range = <0x0 0x1>;
402       compatible = "brcm,bcm2711b0-pcie", ..;
403       max-link-speed = <2>;
404       tot-num-pcie = <1>;
405       linux,pci-domain = <0>;
406       interrupts = <GIC_SPI 148 IRQ_TYPE_LEVEL_HIGH>, ..;
407       interrupt-names = "pcie", "msi";
408       interrupt-map-mask = <0x0 0x0 0x0 0x7>;
409       interrupt-map = <0 0 0 1 &gicv2 GIC_SPI 143 ..>,
410                       <0 0 0 2 &gicv2 GIC_SPI 144 ..>,
411                       <0 0 0 3 &gicv2 GIC_SPI 145 ..>,
412                       <0 0 0 4 &gicv2 GIC_SPI 146 ..>;
413       status = "okay";
414
415       [..]
416     };
417
418     genet: genet@7d580000 {
419       compatible = "brcm,genet-v5";
420       reg = <0x0 0x7d580000 0x10000>;
421       status = "okay";
422       #address-cells = <1>;
423       #size-cells = <1>;
424       interrupts = <GIC_SPI 157 IRQ_TYPE_LEVEL_HIGH>, ..;
425       phy-handle = <&phy1>;
426       phy-mode = "rgmii";
427
428       mdio@e14 {
429         #address-cells = <0>;
430         #size-cells = <1>;
431         compatible = "brcm,genet-mdio-v5";
432         reg = <0xe14 0x8>;
433         reg-names = "mdio";
434
435         phy1: genet-phy@0 {
436           compatible = "ethernet-phy-ieee802.3-c22";
437           max-speed = <1000>;
438           reg = <0x1>;
439           led-modes = <0x00 0x08>;
```

```
        };
      };
    };

    [..]
  };

  v3dbus {
    compatible = "simple-bus";
    #address-cells = <1>;
    #size-cells = <1>;
    ranges = <0x7c500000 0x0 0xfc500000 0x03300000>,
             <0x40000000 0x0 0xff800000 0x00800000>;
    dma-ranges = <0x00000000 0x0 0x00000000 0x3c000000>;

    v3d@7ec04000 {
      compatible = "brcm,2711-v3d";
      reg = <0x7ec00000 0x4000>, ..;
      reg-names = "hub", "core0";
      power-domains = <&pm BCM2835_POWER_DOMAIN_GRAFX_V3D>;
      resets = <&pm BCM2835_RESET_V3D>;
      clocks = <&clocks BCM2835_CLOCK_V3D>;
      interrupts = <GIC_SPI 74 IRQ_TYPE_LEVEL_HIGH>;
      status = "disabled";
    };
  };

  vdd_5v0_reg: fixedregulator_5v0 {
    compatible = "regulator-fixed";
    regulator-name = "5v0";
    regulator-min-microvolt = <5000000>;
    regulator-max-microvolt = <5000000>;
    regulator-always-on;
  };

  sd_io_1v8_reg: sd_io_1v8_reg {
    status = "okay";
    compatible = "regulator-gpio";
    vin-supply = <&vdd_5v0_reg>;
    regulator-name = "vdd-sd-io";
    regulator-min-microvolt = <1800000>;
    regulator-max-microvolt = <3300000>;
    regulator-boot-on;
    regulator-always-on;
    regulator-settling-time-us = <5000>;
```

```
485       gpios = <&expgpio 4 GPIO_ACTIVE_HIGH>;
486       states = <1800000 0x1 3300000 0x0>;
487     };
488
489     sd_vcc_reg: sd_vcc_reg {
490       compatible = "regulator-fixed";
491       regulator-name = "vcc-sd";
492       regulator-min-microvolt = <3300000>;
493       regulator-max-microvolt = <3300000>;
494       regulator-boot-on;
495       enable-active-high;
496       gpio = <&expgpio 6 GPIO_ACTIVE_HIGH>;
497     };
498
499     [..]
500 };
```

Figure A.1: Device tree source code of the Raspberry Pi 4 in standard configuration, with an attached Sense Hat board. The device tree has been aggregated from the different source files and pruned and edited for brevity. The source code is taken from the latest stable Raspbian Linux kernel version 4.19 [14], commit hash `a75a01501330a9be188561b0e9da1da6da372eea` (while writing this thesis the device tree changed on several occasions).

A.2 Raspberry Pi 4 Device Dependency Graph



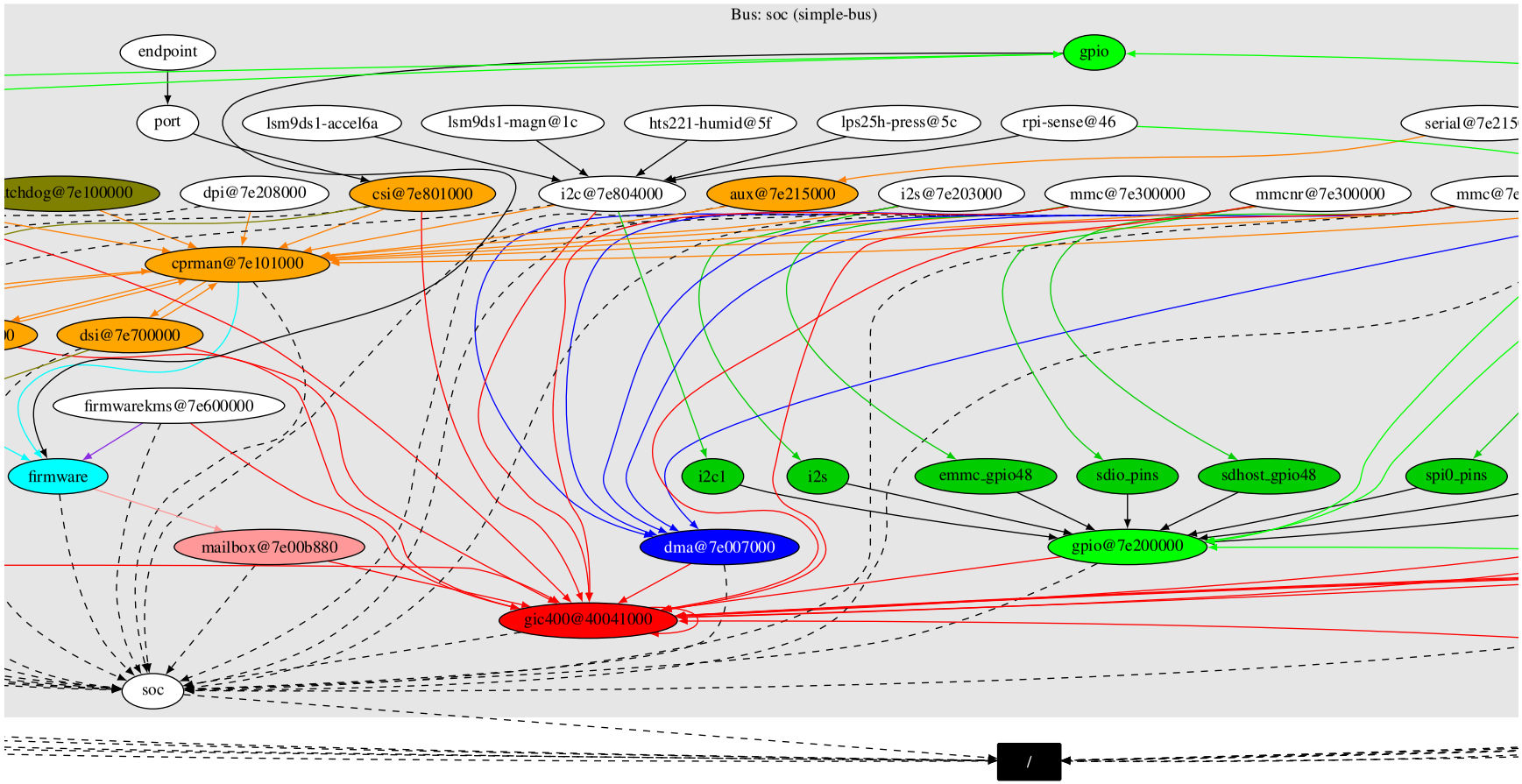
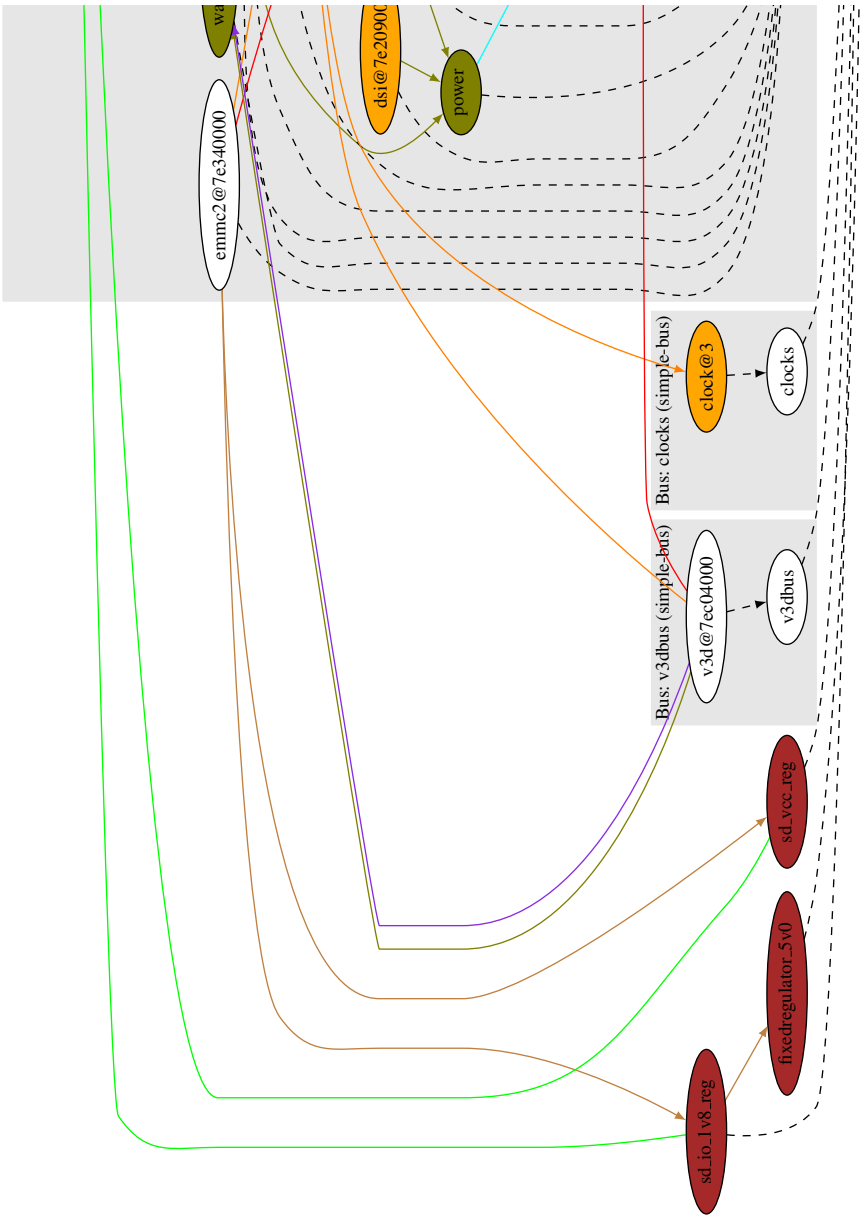


Figure A.2: Visualization of the Raspberry Pi 4 device tree nodes including all dependencies between them. The graph was generated from the device tree described in A.1 using our custom tool `deptree_renderer`. Dashed edges indicate parent-child relationships not regarded as dependencies. The colors describe the node and dependency class :

Root node/parent Unknown dependency Interrupt/MSI controller Mailbox Clock DMA controller
IOMMU GPIO controller Pinconfig controller/config node Firmware Regulator Thermal sensor/zone
PMU or OPP Controller/config node PHY


emmc2@7e340000
dsi@7e209000
power
Bus: clocks (simple-bus)
clock@3
clocks
Bus: v3dbus (simple-bus)
v3d@7ec04000
v3dbus
sd_vcc_reg
fixedregulator_5v0
sd_io_1v8_reg

REFERENCES

[1] D. Goodin. *Record-Breaking DDoS Reportedly Delivered By >145k Hacked Cameras*. Ars Technica. Sept. 2016. URL: https://arstechnica.com/?p=966459 (visited on 04/11/2020).

[2] ByteSnap. "Survey: Biggest Growth Areas for UK Electronics Sector". In: (Mar. 2020). URL: https://www.bytesnap.com/survey-biggest-growth-areas-uk-electronics-sector/.

[3] Unit 42, Palo Alto Networks. *202 Unit 42 IoT Threat Report*. Mar. 2020. URL: https://unit42.paloaltonetworks.com/iot-threat-report-2020/.

[4] L. O'Donnell. *IoT Security Regulation is on the Horizon*. Threatpost. June 2019. URL: https://threatpost.com/iot-security-regulation-horizon/145406/ (visited on 10/30/2019).

[5] B. Schneier. *New IoT Security Regulations*. Nov. 2018. URL: https://www.schneier.com/blog/archives/2018/11/new_iot_securit.html (visited on 10/30/2019).

[6] *The State of California Senate Bill No. 327*. Sept. 2018. URL: https://leginfo.legislature.ca.gov/faces/billNavClient.xhtml?bill_id=201720180SB327.

[7] Samsung. *Refridgerator User Manual - RS27T5200*. Nov. 2019. URL: http://downloadcenter.samsung.com/content/UM/202003/20200304094050460/SBS_RS5300T_DA68-03958J-01_EN_MES_CFR.pdf.

[8] ARM Ltd. *Isolation using Virtualization in the Secure World, Version 1.0*. 2018. URL: https://developer.arm.com/architectures/security-architectures (visited on 04/11/2020).

[9] ARM Ltd. *ARMv8-A Virtualization*. 2017. URL: https://developer.arm.com/architectures/learn-the-architecture/armv8-a-virtualization/single-page (visited on 04/11/2020).

[10] ARM Ltd. *ARM Security Technology - Building a Secure System using TrustZone Technology*. 2009. URL: http://infocenter.arm.com/help/topic/

com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[11] S. Pinto and N. Santos. "Demystifying Arm TrustZone: A Comprehensive Survey". In: *ACM Computing Surveys* 51 (Jan. 2019), pp. 1–36. DOI: 10.1145/3291047.

[12] ARM Ltd. *ARM System Memory Management Unit Architecture Specification*. 2017. URL: https://static.docs.arm.com/ihi0070/b/SMMUv3_architecture_specification_IHI0070B.pdf (visited on 04/11/2020).

[13] O. Schwarz and C. Gehrmann. "Securing DMA through virtualization". In: *2012 Complexity in Engineering (COMPENG). Proceedings*. 2012, pp. 1–6.

[14] *Raspbian Linux Kernel v4.19*. Apr. 2019. URL: https://github.com/raspberrypi/linux/tree/rpi-4.19.y (visited on 04/11/2020).

[15] A. L. Linaro. *Device Tree Specification, Release v0.3*. Feb. 2020. URL: https://github.com/devicetree-org/devicetree-specification/releases/download/v0.3/devicetree-specification-v0.3.pdf (visited on 04/11/2020).

[16] *Linux Kernel Stable Tree v4.19*. Apr. 2020. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/?h=v4.19.115 (visited on 04/11/2020).

[17] N. Semiconductors. *I²C-bus Specification and User Manual, Revision 6*. Apr. 2014. URL: https://www.nxp.com/docs/en/user-guide/UM10204.pdf (visited on 04/11/2020).

[18] J. H. Saltzer and M. D. Schroeder. *The Protection of Information in Computer Systems*. 1975.

[19] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer. "Look Mum, no VM Exits! (Almost)". In: (May 2017).

[20] M. S. Tsirkin, C. Huck, and O. Committee. *Virtual I/O Device (VIRTIO) Version 1.1*. Dec. 2018. URL: https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.html (visited on 04/11/2020).

[21] A. Athalye, A. Belay, M. Kaashoek, R. Morris, and N. Zeldovich. "Notary: a device for secure transaction approval". In: Oct. 2019, pp. 97–113. ISBN: 978-1-4503-6873-5. DOI: 10.1145/3341301.3359661.

[22] S. Pinto, D. Oliveira, J. Pereira, et al. “Towards a Lightweight Embedded Virtualization Architecture Exploiting ARM TrustZone”. In: Sept. 2014. DOI: 10.13140/RG.2.1.3588.1129.

[23] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral. “LTZVisor: TrustZone is the Key”. In: June 2017. DOI: 10.4230/LIPIcs.ECRTS.2017.4.

[24] A. Oliveira, J. Martins, J. Cabral, A. Tavares, and S. Pinto. “TZ- VirtIO: Enabling Standardized Inter-Partition Communication in a Trustzone-Assisted Hypervisor”. In: *2018 IEEE 27th International Symposium on Industrial Electronics (ISIE)*. 2018, pp. 708–713.