Journal of Systems Architecture 119 (2021) 102238

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Self-secured devices: High performance and secure I/O access in TrustZone-based systems☆

Sandro Pinto *, Pedro Machado, Daniel Oliveira, David Cerdeira, Tiago Gomes

Centro ALGORITMI, Universidade do Minho, Portugal

ARTICLE INFO

Keywords:
TrustZone
Security
Virtualization
TEE
Self-secured devices

ABSTRACT

Arm TrustZone is a hardware technology that adds significant value to the ongoing security picture. TrustZone-based systems typically consolidate multiple environments into the same platform, requiring resources to be shared among them. Currently, hardware devices on TrustZone-enabled system-on-chip (SoC) solutions can only be configured as secure or non-secure, which means the dual-world concept of TrustZone is not spread to the inner logic of the devices. The traditional passthrough model dictates that both worlds cannot use the same device concurrently. Furthermore, existing shared device access methods have been proven to cause a negative impact on the overall system in terms of security and performance.

This work introduces the concept of self-secured devices, a novel approach for shared device access in TrustZone-based architectures. This concept extends the TrustZone dual-world model to the device itself, providing a secure and non-secure logical interface in a single device instance. The solution was deployed and evaluated on the LTZVisor, an open-source and lightweight TrustZone-assisted hypervisor. The obtained results are encouraging, demonstrating that our solution requires only a few additional hardware resources when compared with the native device implementation, while providing a secure solution for device sharing.

1. Introduction

For decades, virtualization technology has been efficiently used in partitioning hardware resources between multiple virtual environments [1]. However, with the increase of the embedded system's complexity, there is an added ever-growing need for solutions capable of fulfilling security and real-time requirements [2]. With the advent of the Internet of Things (IoT), security emerged even further as a significant requirement in the embedded systems development. Therefore, ensuring security in such systems is crucial, as they play a key role in safety-critical applications (e.g., aviation, medical, transportation, military), and attacks on cyber–physical systems can potentially cause physical harm [3].

Arm TrustZone [4], Intel Security Guard Extensions (SGX) [5], and Sanctum [6], are examples of security-oriented technologies that promote hardware as the initial root of trust [7,8]. The former is gaining particular attention in the embedded space due to the large presence of Arm processors and microcontrollers in the market. TrustZone technology splits the hardware and software resources into two worlds — the **secure world**, dedicated to the secure processing, and the **normal world** for everything else. A lot of research has been made around TrustZone technology, ranging from efficient and secure virtualization solutions [9–13] to Trusted Execution Environments (TEE) [14–17]. Both approaches, despite targeting different applications with different requirements, consolidate multiple virtual environments in the same platform. Thus, hardware resources are typically shared among them.

In TrustZone-enabled System-on-Chip (SoC) solutions, hardware devices can only be configured as secure or non-secure, meaning that, unlike the physical cores, the device does not enable two execution states simultaneously. Consequently, if both worlds frequently require access to a device through a direct assignment (passthrough) model, this is reflected in a significant increase of the performance penalty. In this case, the device would have to be replicated, which could significantly increase the overall hardware costs.

Currently, shared device access on virtualization- and TrustZone-based architectures can follow different approaches, e.g., (i) proxy task [18], (ii) device emulation [19], (iii) para-virtualization [20,21], (iv) para-TrustZone [9], (v) re-partitioning [22], and (vi) self-virtualizing devices [23,24]. Among all these methods, some bring platform independence and flexibility at the cost of an increased trusted

☆ This work has been supported by FCT -*Fundação para a Ciência e Tecnologia*, Portugal within the R&D Units Project Scope: UIDB/00319/2020.

* Corresponding author.
E-mail addresses: sandro.pinto@dei.uminho.pt (S. Pinto), pedro.machado@dei.uminho.pt (P. Machado), daniel.oliveira@dei.uminho.pt (D. Oliveira), david.cerdeira@dei.uminho.pt (D. Cerdeira), mr.gomes@dei.uminho.pt (T. Gomes).

https://doi.org/10.1016/j.sysarc.2021.102238
Received 4 February 2021; Received in revised form 6 May 2021; Accepted 7 July 2021
Available online 12 July 2021

computing base (TCB) size and execution overhead, while others require considerable engineering efforts or/and hardware costs. Most importantly, some of these approaches disregard fundamental security aspects that may lead devices vulnerable to security attacks [17], which would ultimately cause the device to fail.

In this article, we introduce the concept of self-secure devices in TrustZone-based architectures. Standard TrustZone hardware controllers only implement a single logical and physical interface while offering a set of configuration registers to specify to which world the interface is assigned. In contrast, our self-secured devices approach implements two logical interfaces, simultaneously assigned to the secure and normal worlds. Furthermore, current Arm TrustZone controllers are placed at key points of the main bus, while the self-secure devices solution implements all the logic at the device level at the cost of few additional hardware resources.

The detailed contributions of this article are the following: (1) the extension of the TrustZone concept to devices by separating their hardware logic into a secure and non-secure interface, delivering a new level of security for the overall TrustZone architecture; (2) a quantitative study on the hardware costs introduced by the self-secured devices approach, regarding the device's complexity level; and (3) comparison of the self-secured device approach with existing state-of-the-art methods in terms of engineering effort, memory footprint, performance, and security aspects.

2. Background

2.1. Mixed-criticality systems

An increasingly trend in the design of embedded (and real-time) systems is the integration of components with different levels of criticality onto the same hardware platform. These platforms have been shifting from single- to multi-core architectures, and soon to many-core configurations. Criticality is a designation of the level of assurance against failures, needed for a system component. Thus, a mixed-criticality system (MCS) consolidates both environments and applications with two or more distinct levels [25,26]. For example, in automotive systems, network-connected infotainment is often deployed alongside safety-critical control systems. Most of the embedded systems found in embedded industries, e.g., automotive, avionics, medical, and industrial control, are evolving into mixed-criticality systems in order to meet the market pressure to minimize size, weight, power, and cost (SWaP-C) metrics [26].

2.2. Arm TrustZone

Arm TrustZone consists of hardware security extensions introduced into Arm application processors (Cortex-A) back in 2004 [4,27], and more recently adapted to cover the new generation of Arm microcontrollers (Cortex-M) [13,28,29]. This hardware security extension splits all hardware resources by partitioning a physical processor into two virtual cores: the secure and the normal world. Both worlds are completely hardware isolated and granted uneven privileges, with normal world software prevented from directly accessing secure world resources. TrustZone and TrustZone-M, at a high-level, are identical. However, there are important differences between both processor families, i.e., Cortex-M MCUs are optimized for faster context switch and low-power applications. For the remainder of this section, and under the scope of this work, we will only focus on the description of the TrustZone architecture for Cortex-A processors.

In the TrustZone architecture, the most important change at the processor level consists in the addition of a 33rd bit, used to flag the current processor security state. This bit is accessible through the added Secure Configuration Register (SCR) present in the System Control Co-processor (CP15), and exclusively accessible by the secure world. Secure and normal world partitioning is not only restricted to the processor, but also propagated to other system resources, such as memory, peripherals, and buses. The memory infrastructure can be partitioned into distinct memory regions, which can be configured to be used by both worlds or exclusively by the secure world: (i) the Memory Management Unit (MMU) is banked between secure states, where each world has its unique translation table; (ii) at the cache level, each entry is tagged with the non-secure bit of processor state upon the access and, therefore, entries from both worlds can coexist removing the need for duplication and cache flushing, which accelerates world switching. To enable such memory infrastructure, TrustZone-enabled platforms introduce the TrustZone Address Space Controller (TZASC) and the TrustZone Memory Adapter (TZMA) hardware peripherals. Briefly, TZASC can be used to configure specific memory areas of DRAM as secure or non-secure, whereas TZMA can partition off-chip ROM and SRAM.

To extend TrustZone features to the remaining parts of the SoC, the TrustZone-enabled AMBA Advanced eXtensible Interface (AXI) system bus carries extra control signals, the AWPROT and ARPROT, used to restrict access to the main system bus with three different levels of access protection: (1) normal or privileged, (2) instruction or data, and (3) secure or non-secure. The secure or non-secure accesses are controlled by adding an additional control bit to the system bus, i.e., the non-secure bit, for each of the read and write channels on the main system interconnection. This feature enables TrustZone architecture to secure also system peripherals (e.g., interrupt controllers, timers, and user I/O devices) by using the TrustZone Protection Controller (TZPC), allowing to restrict devices either to secure or normal worlds. The Generic Interrupt Controller (GIC) (both version 2 and version 3, i.e., GICv2 and GICv3) supports the robust management of secure and non-secure interrupts by providing both secure and non-secure prioritized interrupt sources. This prioritization mechanism allows configuring secure interrupts with higher priority than the non-secure interrupts, preventing potential denial-of-service attacks.

2.3. TrustZone-assisted Virtualization

TrustZone technology, although implemented for security purposes, enables a specialized, hardware-assisted, form of system virtualization. With a virtual hardware support for dual world execution, as well as other TrustZone features like memory segmentation, it is possible to provide time and spatial isolation between execution environments. Basically, the non-secure software runs inside a VM whose resources are completely managed and controlled by a hypervisor running in the secure world. TrustZone-assisted virtualization is not particularly considered full-virtualization neither paravirtualization, because, although guest OSes can run without modifications on the normal world side, they need to co-operate regarding the memory map and address space they are using. According to the existing state of the art, TrustZone-assisted virtualization solutions [4] support three types of system configurations: single-guest, dual-guest, and multi-guest. The dual-guest configuration is a perfect fit for embedded mixed-criticality systems, and the one with most works described in the state-of-the-art. LTZVisor [11], discussed in the next subsection, is a great example of a system implementing such configuration.

In the single-guest configuration, the hypervisor runs in the monitor mode, while the guest OS and its applications run in non-secure supervisor and user mode, respectively. In the dual-guest configuration, the hypervisor runs in the monitor mode, and the secure guest OS and its applications run in secure supervisor and user mode, and the normal world hosts the (non-privileged) VM. In the multi-guest configuration, unmodified guest OSes are encapsulated between secure and normal worlds: the active VM runs in the normal world, while the context of inactive VMs is preserved in a secured memory area. This setup requires the hypervisor to effectively handle shared hardware resources, mainly processor registers, memory, caches, and MMU.

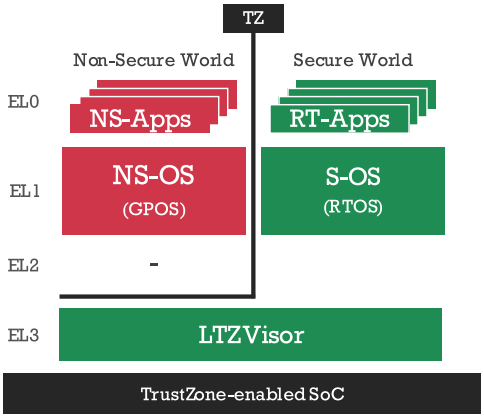


Fig. 1. LTZVisor architecture overview [11].

2.4. LTZVisor

The LTZVisor is a lightweight TrustZone-assisted hypervisor [11]. As depicted by Fig. 1, LTZVisor implements the classic dual-guest model: the secure world hosts the most-privileged virtual machine (VM), i.e., a real-time operating system (RTOS), whereas a general-purpose operating system (GPOS) VM is assigned to the normal world.

The LTZVisor runs in monitor mode, i.e., with the highest privileged processor mode, which allows the hypervisor to have full control of all hardware and software resources. LTZVisor is also responsible for configuring memory, interrupts, and devices assigned to each VM, as well as managing the Virtual Machine Control Block (VMCB) of each VM during partition switches; whenever a VM is about to be executed, the hypervisor saves the CPU state in the VMCB of the running VM and restores the state of the new schedule VM. The secure VM, running on the secure side, runs privileged code that can access or modify any of the non-secure VM resources, such as its memory and associated devices. Therefore, the OS hosted on the secure VM must be aware of its virtualization and it is considered part of the TCB, hence it must keep small TCB size. An RTOS is a perfect candidate to run on the secure side due to its inherent small size and strict time constraints. The non-secure VM, running on the normal world side is ideal to host a GPOS, useful for running human–machine interfaces and containing high-level libraries and other software drivers. The non-secure VM is completely isolated from the privileged VM in the secure world, and whenever an attempt from the normal world to access secure world's resources occurs, an exception to the hypervisor is triggered. Regarding memory, devices, and interrupts, they are configured and assigned to their respective partitions during system initialization and not shared between the VMs.

3. Related work

In this article, we introduce the concept of self-secure devices in TrustZone-based architectures, i.e., we extend the TrustZone dual-world model to the device itself by providing a secure and non-secure logical interface in a single device instance. The concept of self-secured devices is based on the self-virtualizing approach [23,24]. In the remaining of this section we describe existing shared device access approaches for virtualization- and TrustZone-based architectures, namely proxy task [18], para-TrustZone [9], and device re-partitioning [22].

3.1. Self-virtualization technique

A self-virtualizing device [23,24] features hardware logic to provide I/O virtualization functionalities. With such resources, the device is capable of: (i) multiplexing a large number of virtual devices mapped to a single physical device; (ii) managing virtual devices through APIs on the hypervisor; (iii) using APIs for accessing virtual devices and interacting with guest domains; (iv) and taking maximum advantage of the computing power of the hardware platform (e.g., multiple processing cores).

Each device is represented by a virtual interface (VIF), which is accessed from the guest OS through a device driver. The self-virtualized device is responsible for creating, destroying, and managing a VIF. The key task of a self-virtualized device consists of the ability to multiplex/demultiplex several VIFs on a single physical I/O. When a physical device requires to communicate with the processing system, it uses a single communication channel that is then demultiplexed into one of the existing VIF, and data is sent to the respective guest through the VIF's receiving queue.

Self-virtualized devices were designed, from the ground-up, for servers and cloud requirements and materialized in the SRIOV for PCIe [23]. Preeminent solutions for IO virtualization for (embedded) mixed-criticality systems include BlueIO [30] and MCS-IOV [31].

3.2. Proxy task

Proxy Task is the simplest method for sharing devices [18], and it consists of a client task in the secure world OS sending a request, through a well-specified communication channel, to a proxy task running in the normal world OS. These requests are intended to access the device through GPOS's libraries and drivers [12,32]. The drawback of this method lies in the fact that the requests require the collaboration of the non-secure OS, which in the context of TrustZone architectures is a piece of software that cannot be trusted.

3.3. Para-TrustZone

Para-TrustZone [9] is a variation of the para-virtualization technique, which requires the modification of the GPOS driver to send requests to the secure world. This approach uses the Secure Monitor Call (SMC) instruction to perform requests to the secure world. Therefore, the SMC instruction requires kernel privileges and the GPOS driver must be modified to add support for this instruction. This method does not protect the device against the GPOS misbehavior and does not control the request frequency. Therefore, a malicious GPOS can intentionally perform massive SMCs to cause a denial-of-service (DoS) attack and lead the device to block/fail.

3.4. Device re-partitioning

Device re-partitioning allows a device that has been already assigned to a security state to be dynamically re-assigned to another, at run-time [22]. Devices can be configured and re-configured as part of the secure or normal world through the TrustZone Protection Controller (TZPC). As a result, devices can be directly accessed by both the secure and normal worlds, which considerably reduces the performance overhead. Reconfiguration operations (i.e., "PLUG" and "UNPLUG" events) are managed by re-partition managers present in both OSes [32]. The re-partitioning approach can be implemented in a *pure* or *hybrid* form. The disadvantage of this approach is that the device state must be reset when changing security states to guarantee a trustworthy state. This strategy brings huge security implications since a malicious attacker can get full control of the device while it is assigned to the GPOS, making it possible to change and damage its normal operation.

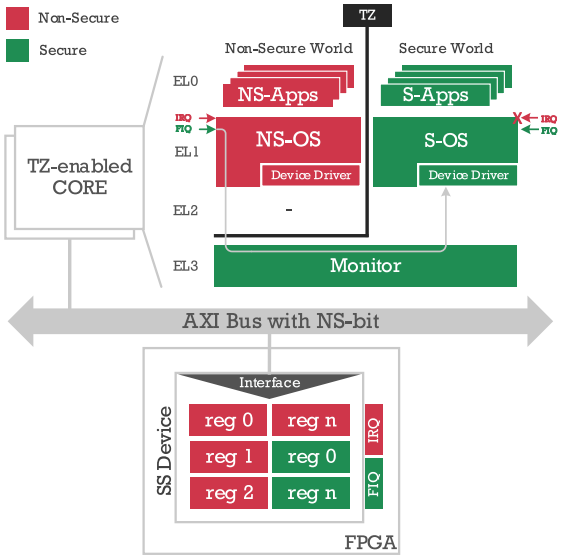


Fig. 2. Self-secured device generic architecture in a platform featuring GICv2.

4. Self-secured devices: Design

TrustZone-based systems are designed to take advantage of a dual world environment, where one world is higher privileged than the other. This concept also extends to devices, where those assigned to the normal world can be accessed by the secure world but the normal world cannot access devices marked as secure. Accesses are performed through the system and peripheral buses, e.g., the Advanced eXtensible Interface (AXI), which are able to propagate the secure state of the CPU through the addition of a control bit on each read and write channel. Therefore, when a request is performed from the normal world (marked by the non-secure bit), the access to sensitive registers and configurations assigned to the secure world is denied. When a device is required by both worlds, current solutions usually follow a time-shared approach or entirely replicate the device's logic. However, time-sharing can be a cumbersome solution and may not be possible to deploy depending on the platform's capabilities, while replicating the entire device incurs on increased hardware costs.

Our approach creates a distinct device interface for each world, securely isolating sensitive registers assigned to each interface (with different banked registers in both interfaces). The solution can grant device access both from the normal and secure worlds simultaneously, without compromising the TrustZone security primitives. This means that the secure world has access to both the secure and non-secure logic interfaces, while the normal world access is restricted to the non-secure logic interface. Moreover, any access from the normal world to the device's configurations can be performed under the supervision of the secure side.

Fig. 2 illustrates the generic architecture of a device following the self-secured approach. The hardware module provides two different accesses, the non-secure and the secure interfaces, which can grant access to different sets of registers, NS REGs and S REGs, respectively. Moreover, the self-secure device must distinguish non-secure from secure interrupts, routing them accordingly from the programmable logic to the processing system. In the GICv2 interrupt controller, secure interrupt sources are routed as Fast Interrupt Request (FIQ), while non-secure interrupts are routed as Interrupt Request (IRQ). If the normal world is executing and an FIQ arises, the execution is transferred to the secure state and the interrupt is handled by the secure-OS (i.e., the RTOS on LTZVisor), avoiding additional overheads in the interrupt latency. On the other hand, when a non-secure IRQ arises and the secure world is executing, the OS behavior is not affected and the interrupt will only be attended as soon as the normal world becomes active again. In the GICv3, the implementation is simpler because interrupts can be marked as secure or non-secure. Furthermore, IRQs can be injected directly to S-EL1, which simplifies interrupt handling on the secure monitor. Therefore, the self-secure device must only provide separate interrupts for each world in a platform featuring GICv3. For the remainder of this article, and due to the focus of our work in the Zynq-7000 SoC architecture, we will drive our implementation for GICv2.

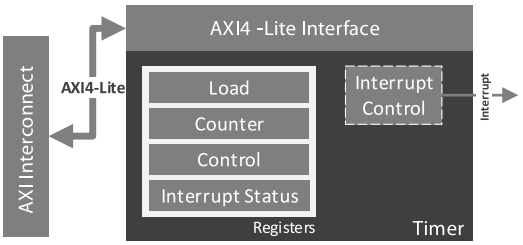


Fig. 3. Block diagram of an AXI timer.

5. Self-secured devices: Implementation

The self-secured device architecture was implemented on a Timer (Fig. 3), and an Universal Asynchronous Receiver/Transmitter (UART) device (Fig. 5). The solution was deployed and tested on a Xilinx Zynq-7000 platform, which features a programmable SoC with a dual-core ARM Cortex-A9 processor along with field-programmable gate array (FPGA) fabric. These devices were selected according to their inner logic complexity, being the UART logic more complex than the Timer, since it requires not only control and status registers, but also data FIFOs and state-machine blocks. This approach will help in understanding the impact of our solution regarding AXI interfaces and registers replication, which will cause a direct impact on hardware costs and overall performance.

5.1. Self-secured device #1: Timer

The Timer device follows the reference implementation of the private timer present in Arm Cortex-A9 MPCore processors [33]. As depicted by Fig. 3, this CPU timer provides four registers: (i) the 32-bit ***Counter*** register, a classic decrementing counter with auto-reload and single-shot modes; (ii) the ***Load*** register, which contains the value copied to the *Counter* register when it decrements down to zero with auto-reload mode enable; (iii) the ***Control*** register, which provides bit assignments for the timer regarding interrupts, auto-reload or single-shot modes, and an 8-bit length clock prescaler; (iv) and the ***Interrupt Status*** register that flags automatically when the Counter register reaches zero.

Fig. 4 depicts the design of the Timer device following the self-secured approach. The device logic is accessed through two separated interfaces (one for each world), preventing the normal world from accessing the most sensitive logic, which could: (i) compromise the counter value of the secure world; (ii) trigger unexpected and unintended interrupts by changing interrupt configurations; (iii) change device configurations that are normally performed at boot time (e.g., prescaler); (iv) or tamper with the device's normal flow by changing the device's secure configuration (e.g., timer mode). To achieve such design some registers, given their atomic nature, must be banked (e.g., *Counter* and *Load* registers), while others can be simply extended (e.g., *Control* and *Interrupt Status* registers), allowing a dual configuration/assignment of functionality bits.

The *Counter* and *Load* registers are entirely duplicated as they compose the functional part of the timer infrastructure. This is not the

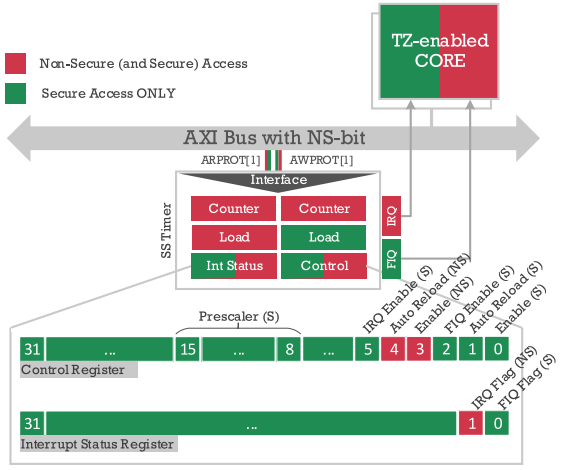


Fig. 4. Self-secured timer hardware architecture.

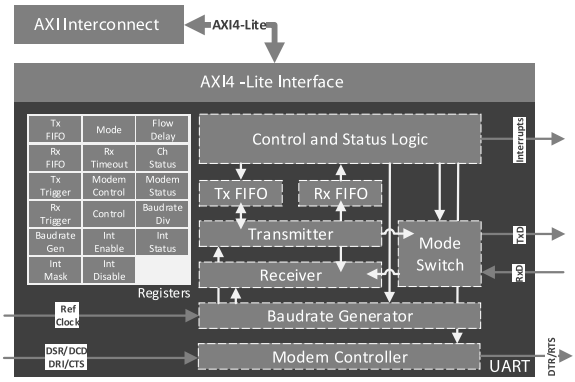


Fig. 5. Block diagram of the UART peripheral.

case of the *Control* and *Interrupt Status* registers, which can be only extended and partially replicated. These registers feature additional bits for the interrupt configuration and event flags, timer activation (i.e., enable bit), and mode configuration (i.e., auto-reload or single-shot). For instance, the prescaler, which influences the overall behavior of counter tick, can only be accessed via the secure world. Likewise, the configuration of interrupts is also exclusively configured by the secure world as well as the FIQ event flag in the *Interrupt Status* register. Furthermore, an additional interrupt source is provided to assign different interrupt mechanisms for each counter overflow. The normal world counter overflow triggers an IRQ, while the secure counter value from the secure register bank, generates an FIQ.

To add these new features, the device AXI register map was extended to include the additional banked registers, i.e., the non-secure *Counter* and *Load* registers. To allow the secure world to access both interfaces and to cope with the required changes in the AXI register addresses, the device driver only requires minor modifications, which include additional functions for reading and writing the *Load* and *Counter* registers of the non-secure interface.

5.2. Self-secured device #2: UART

Fig. 5 illustrates the UART controller, a full-duplex asynchronous receiver and transmitter device used for serial data communication. The device is structured with separate Receiver (Rx) and Transmitter (Tx) data paths, and provides the following features: (i) a programmable baud rate generator; (ii) a receiver and transmitter FIFO with 64 bytes length; (iii) programmable protocol (data size, and stop and parity bits); (iv) error detection for parity, framing, and overrun; (v) line-break and interrupt generation; (vi) multiple TX and RX operation modes; and (vii) modem control signals. In terms of hardware logic, the UART is composed of six modules: (i) control and status logic; (ii) baud rate generator; (iii) transmitter register and Tx FIFO; (iv) receiver register and Rx FIFO; (v) mode switch; and (vi) modem control.

Since the UART device is inherently more complex than the Timer, deploying the self-secured device concept implies a more careful and throughout selection process of which functional, control, and data logic should be accessible from each world. Notwithstanding, this selection process should not compromise the security and functionality of the device, while keeping the hardware costs to the bare minimum. Fig. 6 depicts the self-secured UART device with the secure and non-secure registers. Secure banked registers are exclusively accessed by the secure interface, while non-secure registers can be accessed by both interfaces.

The registers exclusive to the secure interface are explained as follows:

`Control:` This register is assigned to the secure interface since its configuration options (e.g., enable, disable, reset) could potentially compromise the Tx and Rx modules;

`Mode:` This register is responsible for setting the data format (e.g., bit length, parity bit) and selecting the mode of operation. Thus, it is only available in the secure world;

`Interrupt Enable/Disable/Mask:` These registers, usually configured during boot time, are used to control the UART interrupts. Its modification during execution time could trigger unintended interrupts and different execution flows;

`Baud Rate Generator/Divider:` These registers are used to configure the Tx and Rx baud rate speeds. This configuration is typically set at boot time, and, therefore, only accessed by the secure world.

`Receiver Timeout:` This register enables the detection of an idle condition on the Rx data line, issuing an interrupt when the timeout value is reached. It should only be changed by the secure world, otherwise a misconfiguration could trigger unintended timeout interrupts;

`Modem Control:` This register controls the interface with the modem. Only the secure world should be able to change the operation mode and flow control.

`Flow Control Delay:` This register specifies the receiver FIFO level at which the terminal request to send (RTS) signal is asserted/de-asserted. Just like the `Modem Control` register, the `Flow Control Delay` should only be accessible through the secure interface, since it modifies the UART operation mode;

On the other hand, the registers that are accessible and banked between both worlds are the following:

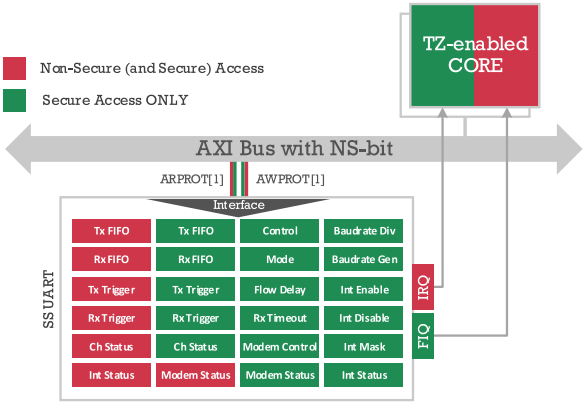


Fig. 6. Self-secured UART hardware architecture.

`Interrupt Status:` This register reflects the state of interrupts. Therefore it must be replicated to the non-secure register bank to enable the normal world to be aware of its own interrupts status;

`Channel` and `Modem Status:` These registers enable the continuous monitoring of the raw unmasked status information of the UART. They must be replicated by both register banks to enable the secure and non-secure FIFO status to be accessible by their respective interfaces;

`Transmit (Tx) FIFO:` This register holds data to be sent by the transmitter. A copy of this register is required in both the secure and non-secure interfaces, preventing secure data from being accessed and/or corrupted;

`Receiver (Rx) FIFO:` This register contains the last received data. A copy of this register is also required in both the secure and non-secure register banks.

`Tx/Rx FIFO Trigger Level:` These registers are used to set the value at which the Tx and Rx FIFO values triggers an interrupt event. These registers must be replicated to be accessible by both the secure and non-secure interfaces.

Following the same interrupt model as in the self-secure Timer device, secure and non-secure interrupts are independent and accessed separately. Therefore, interrupts from the UART in the secure interface are routed as FIQs to the secure world, while interrupts from the non-secure interface are routed as IRQs to the normal world. The AXI address space is also extended with the register banks dedicated to the normal world, filtering the write and read operations according to the extended TrustZone AWPROT and ARPROT protection signals. The transmitter and receiver hardware modules are also slightly modified in the sense that incoming data is stored in the respective FIFO queue, according to their interface type, i.e., secure or non-secure.

Given the higher privilege nature of secure applications using the UART through the secure interface, secure data transmission and reception must be prioritized, which is achieved at the level of transmitter and receiver modules. Secure data transmission prioritization is ensured by only transmitting non-secure data when the secure FIFO is empty. On the other hand, the reception of secure data is ensured by continuously monitoring both secure and non-secure Rx signals. Upon detecting a start bit in the secure interface, the device prioritizes its reception, whether if the module is in idle mode or receiving non-secure data. In the latter, non-secure data will be dumped and the receiver will immediately start receiving the secure data.

6. Evaluation

The evaluation was thoroughly conducted on a Zybo Board, featuring a Zynq-7000 SoC with the processor running at a clock speed of 50 MHz. Since dual-OS configurations for mixed-critical solutions based on Trust-Zone typically deploy a GPOS running in the normal world and an RTOS on the secure world, the self-secure approach was developed in the context of LTZVisor. LTZVisor is a lightweight and TrustZone-based hypervisor that supports a dual-OS configuration, which by design protects the real-time property of the RTOS against GPOS (or other external) attacks to the secure world. The LTZVisor was configured to run the FreeRTOS (version 7.0.2) on the secure VM, and the Linux (2015.4 Xilinx version) on the non-secure VM. The self-secured devices approach, deployed on the Timer and UART devices, was evaluated in terms of (i) number of source lines of code (SLoC) required to modify the native implementation, i.e., hardware modules and software drivers; (ii) memory footprint; (iii) performance; (iv) hardware costs; and (v) security premises. The self-secure device approach is further qualitatively compared with the most relevant state-of-the-art methods for shared devices.

Table 1
Source lines of code (SLoC) for the LTZVisor, RTOS — FreeRTOS (secure world) and GPOS — Linux (normal world).

Devices	Software modules			
	LTZVisor			
	Native	Para-TrustZone	Re-Partitioning	Self-secured
Timer	2259	2302	2259	2265
UART	2259	2334	2259	2265
	RTOS: FreeRTOS			
	Native	Para-TrustZone	Re-Partitioning	Self-secured
Timer	2298	2298	2333	2311
UART	2458	2458	2505	2476
	GPOS: Linux			
	Native	Para-TrustZone	Re-Partitioning	Self-secured
Timer	95	159	131	107
UART	225	305	279	243
	HDL verilog			
	Native	Para-TrustZone	Re-Partitioning	Self-secured
Timer	637	–	–	771
UART	1695	–	–	1970

6.1. Source lines of code (SLoC)

To assess the engineering effort associated with the implementation of the self-secured devices, we have evaluated the impact, in terms of SLoC (summarized in Table 1), required to modify the following components: (1) hardware modules; (2) the LTZVisor; (3) the FreeRTOS; and (4) the GPOS.

Hardware Modifications. Both devices are implemented in Verilog, a hardware description language (HDL) used to model low-level logical systems. For the Timer device, the additional effort to implement the self-secured approach required almost the full replication of the device itself, since the critical logic was independently needed by both the secure and normal worlds. However, the hardware logic to be replicated was simple. For the UART device, and considering that the device complexity is a bit higher than the Timer, the engineering effort had more impact in terms of choosing the critical registers to be replicated than the implementation itself. This is due to the device being classified as a medium complexity device and some registers should not be replicated but exclusively accessed from the secure-interface.

The self-secured Timer device requires 771 SLoC, which represents an increase of nearly 20% when compared with its base implementation (637 SLoC). With a device duplication approach, it would require the full duplication of the number of SLoC. Because the Timer is considered a low-complexity device with few registers to be replicated, the 20% increase is highly related to the additional AXI interface required for the normal world access. For the UART device, the SLoC for the native implementation is 1695, while in the self-secure device approach the SLoC is 1970, which represents an increase of nearly 15%. This can be explained by the fact this device, despite presenting a higher level of complexity, requires less registers and hardware logic to be replicated (most of important registers are exclusively assigned to the secure-interface). From our experience in developing hardware devices and given the obtained results, we can conclude that the overhead caused by modifying a device to our approach decreases meanwhile its complexity increases. Thus, in future implementations of self-secure devices with higher level of complexity, e.g., ethernet interface, it is expected the overhead to be smaller.

LTZVisor Modifications. Table 1 illustrates the number of SLoC required for each software component, using different device sharing methods. Regarding the LTZVisor, the required modifications are related to the security configuration and the routing of the separated interrupts coming from the secure and non-secure device interfaces. While the re-partitioning approach does not require any changes to

Table 2
LTZVisor and FreeRTOS memory footprint.

	Memory footprint in bytes			
LTZVisor image	.text	.data	.bss	Total
Self-secured Timer	51828	468	460440	512736
Timer Re-partitioning	52375	476	460448	513299
Timer Para-TrustZone	52898	468	460456	513822
Self-secured UART	52341	500	460472	513313
UART Re-partitioning	52818	508	460488	513814
UART Para-TrustZone	53582	500	460504	514586

Table 3
Device drivers memory footprint.

	Memory footprint in bytes			
Device drivers	.text	.data	.bss	Total
Native Timer	1764	164	292	2220
Timer Re-partitioning	1764	164	292	2220
Self-secured Timer	1780	164	292	2236
Timer Para-TrustZone	1856	164	300	2324
Native UART	1884	164	548	2496
UART Re-partitioning	1884	164	548	2496
Self-secured UART	1904	164	548	2616
UART Para-TrustZone	1986	164	564	2714

the hypervisor (since the pure method mechanisms are entirely implemented at the OS level), the Para-TrustZone approach is the most intrusive, since is the one that requires more modifications on the LTZVisor. This is a consequence of the hypervisor having to handle every SMC issued by the GPOS driver and the respective device operation based on the passed arguments of every request.

FreeRTOS Modifications. Regarding the FreeRTOS, the para-TrustZone method does not require any modifications, since it only relies on LTZVisor to handle the non-secure requests. However, the self-secure Timer approach requires some minor modifications, that merely consist in providing support for the secure world to perform accesses to both the secure and non-secure device interfaces. In contrast, the additional SLoC in the re-partitioning approach are due to the addition of the required re-partitioning mechanisms, which are responsible for sending the "PLUG" event whenever the RTOS requests the device, and the "UNPLUG" event when it is no longer needed. Furthermore, upon these events, the RTOS must save and restore every device register at each device re-partition, as well as reconfigure the device security according to the world's context switch event.

GPOS Modifications. Regarding the GPOS, the modifications on the self-secured approach only consisted of re-mapping the accesses to the non-secure interface registers and providing support for the non-secure interface interrupt handling. In the re-partitioning method, there are considerable user-level modifications in the GPOS. These modifications are related to the task that runs continuously, checking for upcoming *UNPLUG* and *PLUG* events, which are responsible for unloading and re-loading the device driver, respectively. The Para-TrustZone requires extensive modification to the GPOS driver, as a consequence of replacing the device's operations for SMCs with adequate arguments. This SMCs enable the GPOS to access the secure device through operation requests, sent out directly to the hypervisor.

6.2. Memory footprint

Table 2 contains the LTZVisor and FreeRTOS memory footprint, retrieved with the size tool of the Arm GNU toolchain, for different approaches of the two implemented devices. Because the RTOS is compiled with the hypervisor, all the software stack is considered, as well the hypervisor code, bootloader code, libraries, and FreeRTOS with respective device drivers. The Para-TrustZone, and Re-partitioning approach, introduce the highest memory footprint increase, due to the additional amount of code required to implement this method. Table 3 shows the size of the GPOS device driver for the different implemented approaches. Once again, and for the same reason, Para-TrustZone and re-partitioning introduce the highest memory footprint increase.

6.3. Performance

The performance was evaluated using the Performance Monitor Unit (PMU) module to accurately determine the number of clock cycles required by read/write operations and the latency associated with each device re-partitioning event. Results shown in Table 4 represent the average of one hundred samples and demonstrate a considerable performance overhead introduced by the para-TrustZone method. In contrast, both the self-secure and re-partitioning approaches are similar to the native execution. This is an expected behavior, due to device accesses being performed directly to the hardware. Even though the re-partitioning method does not incur any overhead on read/write accesses, it still introduces considerable device latency (7055 clock cycles) on re-partitioning events, since the FreeRTOS has to wait to reliably use the device.

6.4. Resource utilization

Table 5 summarizes the hardware resources utilization for the Timer and UART devices, relatively to: (1) their native implementations; (2) the self-secured method; and (3) with the device duplication approach. Results were gathered from the Vivado utilization report, which indicates the number of registers, Look-Up Tables (LUTs), I/Os, Global Buffers (BUFGs), Digital Signal Processing (DSP) blocks, and Flip-flops (FFs) required for each implementation. In general, adding more logic to the current design results in a higher utilization of these resources.

Self-secured Timer. For the self-secure and native implementations, the utilization rate of BUFGs and distributed RAM (LUTRAM) remained the same, while the LUTs utilization increased from 3,5% to 4,5%. The FFs utilization had a slight increase from 2% to 2,6%. However, in case of device duplication the following hardware costs are increased: FFs utilization from 2% to 3.2%; LUT utilization from 3,5% to 5,7%; and LUTRAM has a slight increase of 0,05%. The LUTs and FFs utilization had a relative change of 28% and 30%, respectively. In contrast, with device duplication, the LUTs and FFs utilization had a relative change of 60% and 63%, and also a 5% increase of LUTRAM utilization.

Self-Secured UART. With the self-secured approach, the utilization rate of BUFGs, DSP blocks and LUTRAM, remained the same, while the I/O utilization increased from 8% to 10%, the required block RAM (BRAM) from 1,7% to 3,33%, FFs utilization raised from 3,9% to 4,7%, and LUTs increased from 39% to 40,5%. However, if the device is completely replicated, the hardware costs are the following: I/O utilization increased from 8% to 16%; DSP blocks utilization raised from 1,25% to 2,5%; BRAM from 1,7% to 3,33%; FFs utilization increased from 3,9% to 6,9%; LUTRAM had a slight increase from 1% to 1,03%; and the LUTs had the highest increase, from 39% to 74%. Therefore, with the self-secured method, I/Os, BRAM, FFs and LUTs utilization had a relative change of 25%, 95,8%, 20,5% and 3,8%, respectively. However, when duplicating the entire device, the utilization rate of I/Os, DSP blocks, BRAM, FFs, LUTRAM, and LUTs, has a considerable relative increase of 100%, 100%, 95,8%, 77%, 89% and 3%, respectively.

Table 4
Number of clock cycles for write/read device operations and incurred device latency.

	Write	Read	Latency
Native	46	129	0
Self-secured	46	129	0
Re-partitioning	46	129	7055
Para-TrustZone	2233	2125	0

Table 5
Post-implementation hardware costs for the self-secured Timer and UART.

	Timer			UART		
Resource	Native	Self-sec.	Dup.	Native	Self-sec.	Dup.
LUT	610	808	1000	6834	7126	13024
LUTRAM	60	60	62	60	60	62
FF	705	938	1125	1364	1677	2415
BRAM	0	0	0	1	2	2
DSP	0	0	0	1	1	2
IO	0	0	0	8	10	16
BUFG	1	1	1	1	1	1

6.5. Security analysis

Regarding the security properties of the self-secured devices, we have experimentally configured the normal world to access a device through the secure interface. At each attempt, an exception was triggered to the hypervisor, which takes the appropriate actions. Additionally, our approach enforces three fundamental security aspects:

1. Self-secured devices provide confidentiality by means of TrustZone's strong spatial isolation mechanisms. The GPOS cannot access any data allocated on the secure interface nor registers stored in the secure register bank, because the implemented protection mechanism denies any unauthorized access.
2. With the self-secure approach, the device ensures the successful completion of secure operations. Meaning, that an operation performed by the secure world must be completed, regardless of operation requests coming from the non-secure side. This is achieved by restricting non-secure attempts to configure registers that may tamper with the device's normal behavior.
3. The secure world has full control of the device through the secure interface, preventing any action from the normal world that may compromise or inhibit the utilization of the device's resources. Additionally, due to the co-existence of privileged (FIQs) and unprivileged (IRQs) interrupt sources, FIQs belonging to the secure interface can preempt the execution of the GPOS, even when executing an IRQ request.

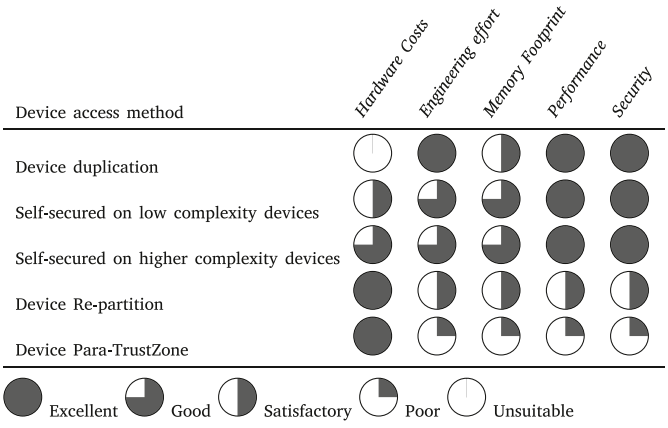
6.6. Qualitative comparison

Each approach to sharing devices in TrustZone has its own strengths and weaknesses. Table 6 provides a side by side qualitative comparison of all the collected evaluations against the self-secure devices approach. For the two software-only methods evaluated, both save the hardware costs. However, device Para-TrustZone introduces overhead caused by the read/write device operation, which requires a significant engineering effort to handle and carry out each SMC. This is due the Para-TrustZone method granting GPOS accesses to the secure device through an SMC. In what concerns the device Re-partitioning method, this overhead is not observed, since it dynamically reassigns the device security in run-time, and once the device is assigned, it allows both OSes to perform read/write accesses. However, the reassignment mechanism implementation requires considerable engineering efforts and introduces performance overhead. Moreover, during the reassignment process period, the device is unusable, incurring a considerable device latency that may be unsuitable for time-critical applications. Most importantly, in both methods, the frequency of GPOS accesses is not limited, which in case of GPOS misbehavior, a large number of requests may be performed, potentially causing the secure device to fail.

From a general point of view, both existing methods lack in terms of security and performance, especially when compared with the approach proposed in this work. Regarding the hardware-assisted implementations, in the device duplication method each guest OS owns a dedicated copy of the device by simply duplicating the entire hardware logic (without any required modifications or additional engineering effort), leaving no margin for security issues or performance degradation. However, and since this solution requires huge hardware costs, it can be unsuitable for embedded systems with a small form factor.

Regarding the self-secured approach, when applied to a low-complexity device, the additional hardware costs are low, requiring minimal protection mechanisms implementations and minimal device driver modifications. This approach achieves native performance and security, guaranteed by TrustZone extensions. When the concept was applied to a medium-complexity device, the incurred hardware costs are still minimal, given the performance-security-hardware spectrum this solution provides.

Table 6
Qualitative comparison between all evaluated device sharing methods.



7. Discussion

In this section, we briefly discuss (i) how to address direct memory access (DMA) support, and (ii) the scalability of our solution.

Self-secure DMA devices. DMA-capable devices were not taken into consideration in the first iteration of this work. Notwithstanding, the generic self-secure device concept and model is broad enough to cover such feature. The concept, in its primitive form, proposes two logic interfaces per physical interface. DMA devices are not much different. We envision DMA support as (i) being integrated into other devices (i.e., DMA is a block embedded into a bigger device) or (ii) as a standalone peripheral. In both cases, the main deviation (or extension) from the already provided infrastructure, is the requirement for the normal world logical interface to not read and write content from secure world addresses. The inclusion of DMA support is one of our main priorities and will be addressed in the near future.

Self-secure devices scalability. We believe the timer and UART are examples of devices with low and middle complexity, respectively. The results presented in Section 6 clearly demonstrated that the higher the complexity of the device, the smaller the hardware overhead, i.e., the additional hardware resources required for the UART device are smaller than those required for the timer. This conclusion seems intuitively valid, as the additional hardware for the additional logical interface

tends to be diluted in the overall logic of the device. Notwithstanding, we want to experimentally validate this conclusion by applying the concept to devices with higher complexity. Thus, we plan to address, as part of future work, network devices, e.g., an Ethernet controller. Naturally, for each device, the self-secure approach involves a trade-off between different metrics, and we not only want to validate the hardware costs, but also understand the overall engineering effort.

8. Conclusions

This work presented a novel approach for shared device access in TrustZone-based architectures, extending the dual-world concept of TrustZone to the inner logic of the device by splitting the device's logic into a secure and non-secure interface. To accomplish this, it was imperative to identify the vulnerable part of the device's logic, that can potentially be exploited, and restrain accesses through the TrustZone extended protection signals, present in the main system bus. This concept was put into practice through the implementation of low- and medium-complexity devices, i.e., a Timer and a UART, in order to evaluate the hardware costs behind such implementations and link them to their complexity level. The results are encouraging, managing to keep the additional hardware costs acceptable for the achieved security enhancements. Increasing the device's complexity, the additional hardware costs are considerable low, when compared to the overall required hardware.

Hereafter, some future work can be pointed: (1) deploy the self-secured approach in a high-complexity device to keep evaluating the trade-off between the hardware costs and the required logic; and (2) extend the self-secure devices to modern microcontrollers that already feature TrustZone-M technology, such as Arm Cortex-M33.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] G. Pék, L. Buttyán, B. Bencsáth, A survey of security issues in hardware virtualization, ACM Comput. Surv. 45 (2013) 40:1–40:34.

[2] T. Shimada, T. Yashiro, N. Koshizuka, K. Sakamura, A real-time hypervisor for embedded systems with hardware virtualization support, in: 2015 TRON Symposium, TRONSHOW, 2014, pp. 1–7.

[3] R. Langner, Stuxnet: Dissecting a cyberwarfare weapon, IEEE Secur. Priv. 9 (2011).

[4] S. Pinto, N. Santos, Demystifying arm trustzone: A comprehensive survey, ACM Comput. Surv. 51 (6) (2019).

[5] V. Costan, S. Devadas, Intel SGX explained, 2016, Cryptology ePrint Archive, Report 2016/086.

[6] V. Costan, I. Lebedev, S. Devadas, Sanctum: Minimal hardware extensions for strong software isolation, in: 25th USENIX Security Symposium (USENIX Security 16), USENIX Association, Austin, TX, 2016.

[7] P. Nasahl, R. Schilling, M. Werner, S. Mangard, HECTOR-V: A heterogeneous CPU architecture for a secure RISC-V execution environment, 2020, URL arXiv: 2009.05262.

[8] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, E. Stapf, CURE: A security architecture with customizable and resilient enclaves, in: 30th USENIX Security Symposium, USENIX Security 21, USENIX Association, Vancouver, B.C., 2021.

[9] T. Frenzel, A. Lackorzynski, A. Warg, H. Härtig, ARM TrustZone as a virtualization technique in embedded systems, in: Proceedings of Twelfth Real-Time Linux Workshop, Nairobi, Kenya, 2010.

[10] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, H. Guan, vTZ: Virtualizing ARM trustzone, in: Proceedings of the 26th USENIX Conference on Security Symposium, SEC'17, 2017, pp. 541–556.

[11] S. Pinto, J. Pereira, T. Gomes, A. Tavares, J. Cabral, LTZVisor: TrustZone is the key, in: 29th Euromicro Conference on Real-Time Systems, ECRTS 2017, in: Leibniz International Proceedings in Informatics, LIPIcs, vol. 76, 2017, pp. 4:1–4:22.

[12] A. Oliveira, J. Martins, J. Cabral, A. Tavares, 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.

[13] S. Pinto, H. Araujo, D. Oliveira, J. Martins, A. Tavares, Virtualization on TrustZone-enabled microcontrollers? Voilà! in: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, 2019.

[14] A.M. Azab, K. Swidowski, R. Bhutkar, J. Ma, W. Shen, R. Wang, P. Ning, SKEE: A lightweight secure kernel-level execution environment for ARM, in: Proceedings of the Network and Distributed System Security Symposium, 2016.

[15] S. Pinto, T. Gomes, J. Pereira, J. Cabral, A. Tavares, IIoTEED: An enhanced, trusted execution environment for industrial IoT edge devices, IEEE Internet Comput. 21 (2017) 40–47.

[16] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, E. Stapf, SANCTUARY: ARMing TrustZone with user-space enclaves, in: Network and Distributed Systems Security, NDSS Symposium, 2019.

[17] D. Cerdeira, N. Santos, P. Fonseca, S. Pinto, SoK: Understanding the prevailing security vulnerabilities in trustzone-assisted TEE systems, in: 2020 IEEE Symposium on Security and Privacy, SP, IEEE Computer Society, Los Alamitos, CA, USA, 2020, pp. 636–652.

[18] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, T. Li, Building trusted path on untrusted device drivers for mobile devices, in: Proceedings of 5th Asia-Pacific Workshop on Systems, APSys '14, 2014.

[19] J. Sugerman, G. Venkitachalam, B.-H. Lim, Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor, in: Proceedings of the General Track: 2001 USENIX Annual Technical Conference, 2001, pp. 1–14.

[20] S. Kim, C. Lee, M. Jeon, H. Kwon, H. Lee, C. Yoo, Secure device access for automotive software, in: 2013 International Conference on Connected Vehicles and Expo, ICCVE 2013 - Proceedings, 2013, pp. 177–181.

[21] S.A. Babu, M.J. Hareesh, J.P. Martin, S. Cherian, Y. Sastri, System performance evaluation of para virtualization, container virtualization, and full virtualization using Xen, OpenVZ, and XenServer, in: 2014 Fourth International Conference on Advances in Computing and Communications, 2014, pp. 247–250.

[22] D. Sangorrín, S. Honda, H. Takada, Reliable device sharing mechanisms for Dual-OS embedded trusted computing, in: TRUST'12 Proceedings of the 5th International Conference on Trust and Trustworthy, vol. 7344, 2012, pp. 74–91.

[23] H. Raj, K. Schwan, High performance and scalable I/O virtualization via self-virtualized devices, in: Proceedings of the 16th International Symposium on High Performance Distributed Computing, HPDC '07, 2007.

[24] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A.L. Cox, W. Zwaenepoel, Concurrent direct network access for virtual machine monitors, in: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture, HPCA '07, 2007.

[25] A. Burns, R.I. Davis, A survey of research into mixed criticality systems, ACM Comput. Surv. 50 (6) (2017).

[26] J. Martins, A. Tavares, M. Solieri, M. Bertogna, S. Pinto, Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems, in: M. Bertogna, F. Terraneo (Eds.), Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020), in: OpenAccess Series in Informatics (OASIcs), vol. 77, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2020, pp. 3:1–3:14.

[27] ARM, ARM® Cortex® -a Portfolio ARMv8-a, Tech. Rep., ARM, 2016, pp. 1–6.

[28] T. Nyman, J.-E. Ekberg, L. Davi, N. Asokan, CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers, in: International Symposium on Research in Attacks, Intrusions, and Defenses, Springer International Publishing, 2017, pp. 259–284.

[29] N. Asokan, T. Nyman, N. Rattanavipanon, A. Sadeghi, G. Tsudik, ASSURED: Architecture for secure software update of realistic embedded devices, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 37 (11) (2018) 2290–2300.

[30] Z. Jiang, N. Audsley, P. Dong, BlueIO: A scalable real-time hardware I/O virtualization system for many-core embedded systems, ACM Trans. Embed. Comput. Syst. 18 (3) (2019).

[31] Z. Jiang, N. Audsley, P. Dong, N. Guan, X. Dai, L. Wei, MCS-IOV: Real-time I/O virtualization for mixed-criticality systems, in: 2019 IEEE Real-Time Systems Symposium, RTSS, 2019, pp. 326–338.

[32] D. Sangorrín, S. Honda, H. Takada, Reliable and efficient dual-os communications for real-time embedded virtualization, Inf. Media Technol. 8 (1) (2013) 1–17.

[33] ARM, Cortex -A9 MPCore: Technical Reference Manual, Tech. Rep., ARM, 2012.