

École Polytechnique Fédérale de Lausanne

On the Enforcement of Attestable Availability Guarantees for Arm-based Industrial Multi-Tenant Real-Time Systems

by Edouard Michelin

Master Thesis

Approved by the Examining Committee:

Prof. George Candea
Thesis Advisor

Dr. Christian Göttel, Dr. David Kozhaya
Thesis Supervisors

EPFL IC IINFCOM DSLAB
INN 330 (Bâtiment INN)
Station 14
CH-1015 Lausanne

ABB Corporate Research Center
Industrial Software Systems
Segelhofstrasse 1K
CH-5405 Baden-Dättwil

August 29, 2025

The ultimate inspiration is the deadline.
— Nolan Bushnell

Dedicated to my parents, Christophe and Maryline, and to my brother, Charles.

Acknowledgments

I would like to begin by thanking my family for encouraging me to pursue my studies and for supporting me along the way. I would not be where I am today without them.

I am grateful to my supervisors at ABB, Dr. Christian Göttel and Dr. David Kozhaya, for the opportunity to carry out this thesis with them, as well as for their valuable guidance and advice throughout the project.

I also thank my EPFL supervisor, Prof. George Candea, for his support during my research and for the opportunities he has offered me during my studies.

Baden, August 29, 2025 Edouard Michelin

Abstract

With the rise of Industry 4.0, industries increasingly rely on digital technologies to enable automation, remote control, as well as other real-time services. The integration of sensors, smart devices, and high-speed connectivity is transforming traditional industrial systems into interconnected cyber-physical infrastructures. The convergence of information technology (IT) and operational technology (OT) offers significant operational benefits but also introduces critical security challenges due to fundamental differences between the two domains. OT systems are constrained by strict timing requirements and conservative standards, while IT requires frequent, precise security updates. To remediate the ripples created when bringing these two worlds together, we propose a novel security architecture that bridges remote attestation of execution with guarantees of real-time availability for industrial control systems (ICS). Leveraging commodity Arm TrustZone platforms, our design enforces user-defined policies, and ensures predictable and verifiable operation. We implement a prototype that demonstrates support for high-frequency real-time workloads (period shorter than 10μs) with low scheduling latency (as low as 6μs), preserving confidentiality and integrity while providing robust attestation, marking a practical path for the secure digital transformation of industrial systems.

Résumé

Avec l'essor de l'Industrie 4.0, les industries s'appuient de plus en plus sur les technologies numériques afin de permettre l'automatisation, le contrôle à distance ainsi que d'autres services en temps réel. L'intégration de capteurs, d'appareils intelligents et de connexions à haute vitesse transforme les systèmes industriels traditionnels en infrastructures cyber-physiques interconnectées. La convergence des technologies de l'information (IT) et des technologies opérationnelles (OT) offre des avantages opérationnels significatifs, mais introduit également des défis de sécurité critiques en raison des différences fondamentales entre ces deux domaines. Les systèmes OT sont soumis à des contraintes strictes en matière de temps d'exécution et à des normes conservatrices, tandis que les systèmes IT nécessitent des mises à jour de sécurité fréquentes et précises. Afin de remédier aux problèmes générés par cette convergence, nous proposons une nouvelle architecture de sécurité qui associe l'attestation d'exécution à distance à des garanties de disponibilité en temps réel pour des systèmes de contrôle industriel (ICS). En tirant parti des plateformes Arm TrustZone grand public, notre conception applique des politiques définies par l'utilisateur et assure un fonctionnement prévisible et vérifiable. Nous mettons en œuvre un prototype qui démontre la prise en charge de tâches temps réel à haute fréquence (période inférieure à 10μs) avec une faible latence de planification (aussi basse que 6μs), tout en préservant la confidentialité, l'intégrité et en offrant une attestation robuste, ouvrant ainsi une voie concrète vers une transformation numérique sécurisée des systèmes industriels.

Contents

Chapter 1

Introduction

With the ongoing fourth industrial revolution, industries rely more than ever on digital technologies to drive automation, efficiency, and new services. The integration of sensing capabilities, smart devices, and high-speed connectivity is transforming traditional mechanical industrial systems into cyber-physical ones. This trend, often summarized under the term *Industry 4.0*, promises significant benefits such as predictive maintenance, adaptive production, and real-time monitoring. At the same time, it increases the reliance on interconnected infrastructures and broadens the potential attack surface. The once clear line between Information Technology (IT) and Operational Technology (OT) is becoming increasingly blurred as both domains converge.

While IT has a long tradition of rapidly adopting new security and networking solutions, OT environments tend to be more conservative due to regulatory requirements, standardization, and the high costs associated with downtime or recertification. As a result, OT standards are updated infrequently, and standardization bodies often struggle to reach consensus on specific implementation details, producing broad standards that lack precision. In contrast, IT security must be continuously updated and implemented with precision, as malicious actors will exploit any gaps in the architecture. Historically, OT has been viewed as a constrained environment, with access limited to personnel working in the field, whereas IT spans a much larger scope, resulting in an exponentially larger attack surface. Bringing IT and OT together introduces a wide range of potential vulnerabilities, with numerous intrusion vectors, into a world that, until the advent of Industry 4.0, largely operated in compartmentalized environments with limited focus on security. IT resources, being widely accessible and often open-source, benefit from a large community that helps identify and fix implementation bugs. OT, on the other hand, has relied on specialized equipment and a more closed-source approach, which has historically limited its security capabilities, particularly given the operational constraints and strict timing requirements of industrial control systems.

The convergence of IT and OT, while offering significant operational benefits, also introduces critical vulnerabilities because of the fundamental differences between the two domains. In this

work, we address the ripples created by bridging these two worlds. We propose a security architecture that suits both environments, one that preserves the real-time reliability and deterministic behavior of industrial systems, while also offering new security perspectives to the OT world to defend against modern threats. More specifically, we introduce a hardware-software co-design that provides attestable availability guarantees.

In recent years, research has addressed availability guarantees for industrial applications [4, 51, 85, 86, 89], as well as proof of execution [23, 54, 56, 87]. To the best of our knowledge, we are the first to bridge the gap between these security properties. We build on the widely available Arm TrustZone [15], analyze the requirements for such an architecture, and propose a design that can serve as a blueprint for future industrial environments. To validate our approach, we implement a prototype on a TrustZone-enabled development kit and evaluate its performance to assess both applicability and limitations. Our evaluation shows that the proposed design can be realized on commercially available hardware and applied to representative workloads. The results suggest that our architecture can preserve real-time constraints while offering combined security properties, indicating a viable path for the secure digital transformation of industrial systems.

In summary, we make the following contributions:

- A novel security architecture that bridges attestation of execution with guarantees of real-time availability, even in the presence of a fully compromised normal world OS present from system startup.
- An Availability Monitor that enforces user-defined policies to maintain real-time availability, and an Attestation Engine that produces verifiable evidence of correct scheduling, execution, and configuration.
- Commodity-ready deployment by leveraging resource-efficient Arm multiprocessor platforms, enabling predictable and verifiable trust for critical industrial infrastructures.
- A prototype implementation and evaluation showing that our architecture supports high-frequency real-time execution (up to 110 kHz) with low scheduling latency (as low as 6 μs), while preserving confidentiality and integrity of proprietary code and data.

Chapter 2

Background

2.1 Industrial Control Systems

Industrial Control Systems (ICS) are at the core of Operational Technology (OT). Their role is to ensure that industrial processes follow their expected behavior. They do so by repeatedly collecting data, received from one or more sensors, performing validation operations, and taking action based on the outcome of the verification. This verification procedure, referred to as control loop or control cycle, is run periodically and is most often mission-critical. Consequently, the control loops in an ICS have high availability requirements, and must react to inputs from their environment before a faulty process can have an impact on subsequent operations. This imposed time limit is referred to as deadline, and the tasks that need to complete in such given time constraints, i.e., before reaching a deadline, are called real-time tasks.

2.1.1 Real-Time Systems

A real-time system comprises a set of cores (here sometimes referred to as computational resources), a set of shared resources (which we differentiate from computational resources by referring to them as non-computational resources, devices, or peripherals), and a set of real-time tasks [27]. Contrary to common assumptions, real-time systems impose timing constraints related to a deadline, and need not necessarily be fast or instantaneous. When instances of a task, called jobs, must always complete before reaching their deadline, we say that the timing constraint is hard. Whereas it is said to be soft when the deadline can be missed on some occasion, with low probability, and without having a direct negative impact. In case the tasks of a real-time system have a hard deadline, that system is called a hard real-time system, and we say that the tasks of such a system are critical. Depending on the way they are released (i.e., scheduled), tasks can be periodic, sporadic, or aperiodic. An aperiodic task is scheduled at unpredictable intervals. On the opposite side of the spectrum,

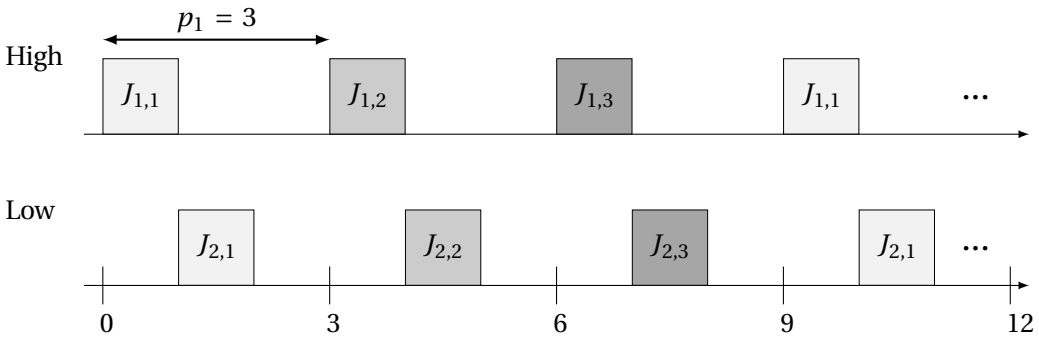


Figure 2.1: Scheduling of two tasks $T_1 = (3,1,1,\emptyset)$ and $T_2 = (3,1,1,\emptyset)$ with $P(T_1) > P(T_2)$, each of which has three jobs, i.e., $J(T_1) = \{ J_{1,1}, J_{1,2}, J_{1,3} \}$ and $J(T_2) = \{ J_{2,1}, J_{2,2}, J_{2,3} \}$.

periodic tasks are scheduled at strict, predefined cycles. Lastly, a sporadic task is an aperiodic task with a deadline. In ICS the focus is on periodic tasks, as control loops are released at regular intervals and must complete before their deadline.

Let $\mathscr{C} = \{ C_i \mid i = [1,n],\ n \in \mathbb{N} \}$ be the set of cores of the system, $\mathscr{T} = \{ T_i \mid i = [1,n],\ n \in \mathbb{N} \}$ be the set of tasks of the system, and $\mathscr{R} = \{ R_i \mid i = [1,n],\ n \in \mathbb{N} \}$ be the set of shared resources available on the system. Formally, we characterize the periodic tasks of a system as a 6-tuple $(\phi,\ p,\ e,\ D,\ c,\ R)$, where ϕ is the release time of its first job on the system, called its phase, p is its period, that is, the time interval between the release of successive jobs, e is the execution time of the task, often defined as the Worst Case Execution Time (WCET) of all its jobs, D is the minimum relative deadline of its jobs, $c \in \mathscr{C}$ is the core on which the task will be scheduled and run, and $R \subset \mathscr{R}$ is the set of resources the task accesses. In order to simplify this definition, for all tasks we say that the phase is defined as the starting time of the system, i.e., $\phi = 0$, and the relative deadline of the jobs is equal to the period, that is, $D = p$, so that the tasks $T_i \in \mathscr{T}$ in the system are defined as the 4-tuple $(p_i,\ e_i,\ c_i,\ R_i)$. The utilization u_i of a task $T_i \in \mathscr{T}$, given by $U(T_i) = u_i$, is computed as $u_i = e_i / p_i$, and provides a measure of the proportion of computing resources consumed by the task. We denote $J(T_i)$ the set of jobs of the task $T_i \in \mathscr{T}$. The jobs $J_{i,j} \in J(T_i),\ T_i \in \mathscr{T},\ j \in [1, |J(T_i)|]$ of a task $T_i \in \mathscr{T}$ are scheduled in order, with the interval of time between two consecutive releases being their period. In other words, all jobs $J_{i,j},\ j > 1$ of a task T_i are scheduled and must complete in the time frame $[(j-1) \cdot p_i,\ j \cdot p_i]$ with time frame of $J_{i,1}$ being $[0,\ p_i]$. The order in which the jobs of different tasks are scheduled depends on the priority of their task. Different algorithms are used to compute the priority of tasks, which can either be fixed, or dynamic. Two well-known examples of such algorithms are the fixed priority Rate Monotonic (RM) [49] algorithm, which gives higher priority to tasks with a shorter period, and the dynamic priority Earliest Deadline First (EDF) [21], which, every time the scheduler has to take a scheduling decision, gives precedence to the task closest to its deadline. We denote $P(T_i)$ the priority assigned to task $T_i \in \mathscr{T}$, and illustrate the scheduling of two tasks $T_1 = (3,1,1,\emptyset)$ and $T_2 = (3,1,1,\emptyset)$ with $P(T_1) > P(T_2)$, each of which has three jobs, i.e., $J(T_1) = \{ J_{1,1}, J_{1,2}, J_{1,3} \}$ and $J(T_2) = \{ J_{2,1}, J_{2,2}, J_{2,3} \}$ in Figure 2.1.

Periodic and aperiodic tasks can coexist on a hard real-time system. In that case, the objective of the task scheduler is that periodic tasks do not miss their deadline, while minimizing the response time of aperiodic tasks. The simplest way to fulfill this mission, when the response time of the aperiodic tasks is not critical, is to treat them as background tasks [47, 88]. In background scheduling, aperiodic tasks are assigned the lowest priority on the system, which as a result do not impact the schedulability of periodic tasks in any way. However, it presents the disadvantage of increasing the response time of aperiodic tasks as the total utilization of the system by critical tasks, denoted $\mathscr{U} = \sum_{T_i \in \mathscr{T}} U(T_i)$, increases. There exist more advanced techniques that provide a faster response time to aperiodic, which leverage a periodic server, such as a polling server or a deferrable server [70]. However, the faster response times come at the cost of increased scheduler complexity.

2.1.2 Scheduling of Shared Resources

Traditional schedulers, used for instance in consumer personal computer systems, primarily focus on managing the availability of computational resources. Their role is to balance the execution of processes across CPU cores according to some strategy, for which the objectives are often to provide overall system performance, fairness towards all processes, and good user experience. These objectives differ significantly from those of real-time schedulers, making them unsuitable for real-time tasks [74]. Real-time constraints require the scheduling of both computational resources and non-computational resources, such as connected devices (e.g., network card, sensors, actuators, etc.), which need to be obtained by real-time tasks in a deterministic and timely manner, that is, within a strict timing constraint. Should a required sensor or actuator become unavailable and cause a job to miss its deadline, the entire protection function (i.e., control loop) could become worthless, which could have severe consequences in a safety-critical real-time system. An example of such a scenario is the incapacity of a control loop to obtain up-to-date data from a proximity sensor in industrial robotics, where articulated robots [1] can lift heavy payloads, such as car parts. These sensors serve to cut system power when two moving parts come dangerously close, preventing costly collisions.

The scheduling of shared devices on a system brings new challenges, potentially creating scheduling problems. Two well-known scheduling problems are deadlocks, and priority inversions, in which a lower priority task holding the lock on a shared resource prevents a higher priority task from acquiring the lock and executing. This issue becomes especially important when tasks that do not use the shared resource, but have a priority higher than the task holding the lock and lower than the blocked task, preempt the execution of the lock owner, thereby delaying its completion and the release of the lock needed by the higher priority task. As depicted in Figure 2.2, this scenario leads to an unbounded priority inversion, which can ultimately cause system-wide failure. The situation was observed during the Mars Pathfinder mission, causing total system resets in the spacecraft that jeopardized the mission's success [72]. Fortunately, a wide range of suspension- and spin-based resource sharing protocols [20, 27, 32, 62, 63] have been proposed for hard real-time systems over the

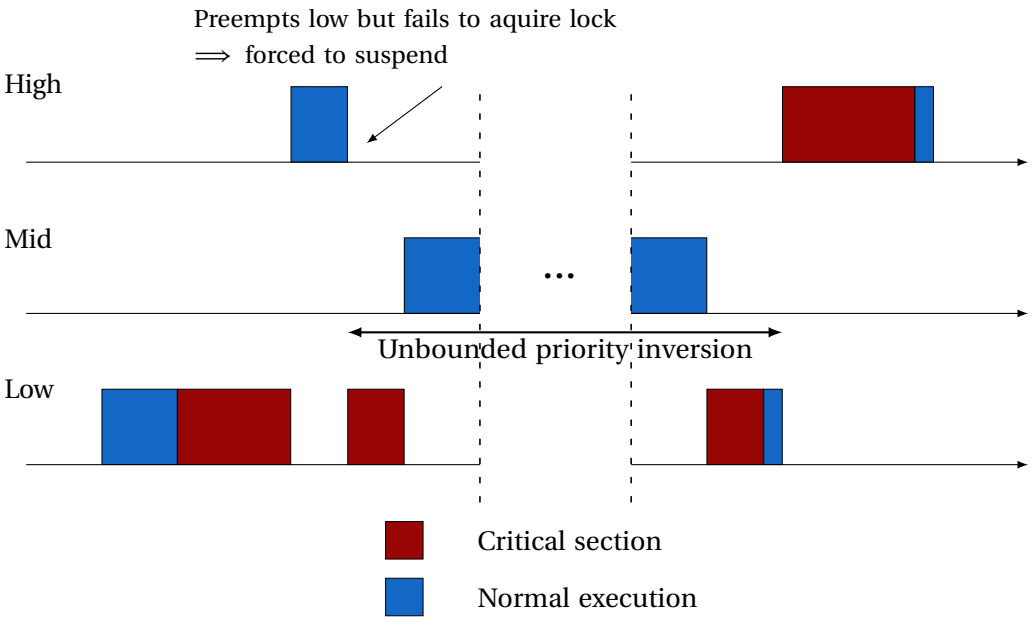


Figure 2.2: Example of an unbounded priority inversion when a high priority task is blocked by a low priority task locking a shared resource, and that task is preempted by a medium priority one.

years. Among these protocols, the Multiprocessor Priority Ceiling Protocol (M-PCP or MPCP) [62] is a suspension-based protocol for partitioned fixed-priority scheduling, which prevents deadlocks and bounds priority inversion to the duration of the interval in which a resource holds the lock on a device, known as a critical section [68]. This is achieved by lifting the effective priority of tasks holding a lock on a resource above that of all other tasks in the system. As a result, tasks in a critical section cannot be preempted. Although quite simple to implement, a limitation of this protocol is that it does not support nested critical sections, since a deadlock could occur when two tasks simultaneously compete for two or more common resources.

2.1.3 Availability of Resources

In trusted computing, availability is often overlooked in favor of confidentiality, integrity, and other security properties. In fact, most trusted operating systems [5, 61, 82] delegate the scheduling of applications to the untrusted OS [45], as (1) moving the scheduling logic to the trusted domain would significantly increase the size and complexity of the TCB, and (2) they are often designed as companion OSes. For instance, the Completely Fair Scheduler (CFS) in Linux v6.14 [35] is approximately 13,700 lines of code. Although the aftermath of disrupting the availability of a system such as a personal computer or a phone is in most cases not dramatic, the conclusions are different when dealing with critical real-time systems, such as industrial plants, avionics, or modern cars [39]. In this project, our effort goes into ensuring the availability of resources needed by real-time tasks.

In other words, when a task needs to run, we must ensure that all the resources it requires are available, so that it can perform its operations properly and within delays. However, such guarantees cannot be provided if the very component that manages the allocation of resources on the system is completely untrusted, and, as such, at least a minimal subset of scheduling decisions has to be trusted to enforce policies.

2.1.4 Multi-Tenancy

Whether in our car, on the plane, in the substations that deliver power to our house, in medical devices such as pacemakers, or even in our washing machine, the vast majority of real-time systems that we encounter everyday are closed systems. A closed system is a system in which all software and hardware components are known and fixed a priori, and for which changes are not possible at runtime. This usually allows system designers to make assumptions about security, schedulability of resources, as well as other aspects of the development that will facilitate the design and implementation. However, this lack of flexibility can rapidly become a limitation, particularly in industrial contexts, where a customer could decide to use the system offered by company $\mathscr{A}$ for controlling some aspect of their plant, while also needing at a later point in time the software solution of company $\mathscr{B}$ for monitoring other on-site events, which the customer should be able to setup dynamically at runtime, without having to reconfigure the entire machine. Unfortunately, this ability to have a multi-tenant, open system is not free, as many assumptions that were possible in a closed system no longer hold.

2.2 Trusted Computing on Arm Platforms

Arm AArch64 processors employ varying levels of privileges, known as Exception Levels (EL) [11], similar to the concept of rings on x86 processors. These privilege levels range from the least privileged EL0, usually reserved to user-level applications, to the most privileged EL3, typically reserved for the boot loader and security monitor. They enforce a hierarchical, horizontal isolation, which can show insufficient to properly isolate software components at the same level of privilege.

To address this issue, Arm A-profile processors enabled by the Security Extensions employ an additional isolation mechanism named TrustZone [15], which allows confidential computing on the Arm platform. More specifically, TrustZone allows system developers to partition platform resources (e.g., peripherals, memory) into two distinct regions commonly referred to as worlds, which are environments isolated from each other at the hardware level, following the access control matrix specified in Table 2.1. The TrustZone architecture, shown in Figure 2.3, introduces two worlds running alongside each other. The so-called secure world hosts the confidential environment and is used for security-sensitive operations. It comprises a trusted OS (TOS) with a minimal set of functionalities and trusted applications (TA). The secure OS is limited in the sense that it does not, for

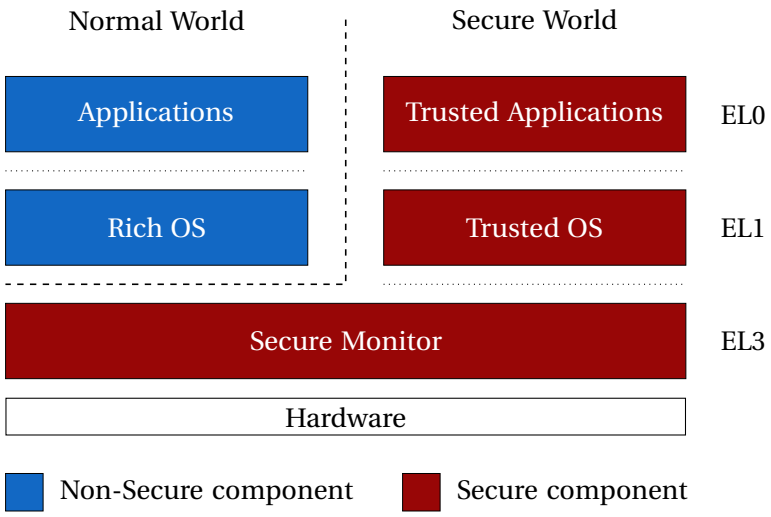


Figure 2.3: Schematic architecture of a TrustZone-enabled system. The thick dashed line represents the worlds boundary. The thin dotted lines represent the in-world isolation between ELs. For simplicity, EL2 is not shown.

	Secure Region	**Normal Region**
Secure Context	✓	✓
Normal Context	✗	✓

Table 2.1: Access Control Matrix of the TrustZone architecture. When a CPU core is executing in secure mode, it has access to both secure and normal resources, whereas when executing in normal mode, it only has access to normal resources, i.e., access to secure resources is forbidden.

instance, implement device drivers or scheduling capabilities (passive OS) making it dependent on its feature-rich counterpart environment. The normal world, also referred to as non-secure world, hosts the untrusted, feature-rich part of the system, the so-called Rich Execution Environment (REE). The scheduling of TAs is initiated by client applications (CA) in the normal world when a sensitive service is requested. The switch from normal world to secure world is then performed via a conduit method, the `smc` (Secure Monitor Call) privileged instruction [16], which preempts execution and traps to the secure monitor (SM) that forwards the request to the TOS. This context switch is only performed on the core executing the `smc` instruction, and does not alter the state of the other processors on the platform, which can run in any of the two worlds independently. Should a scheduled TA require access to an external device (e.g., network card or hard drive), it will send a request back to the normal world – implementing the necessary drivers – which will execute the task on its behalf.

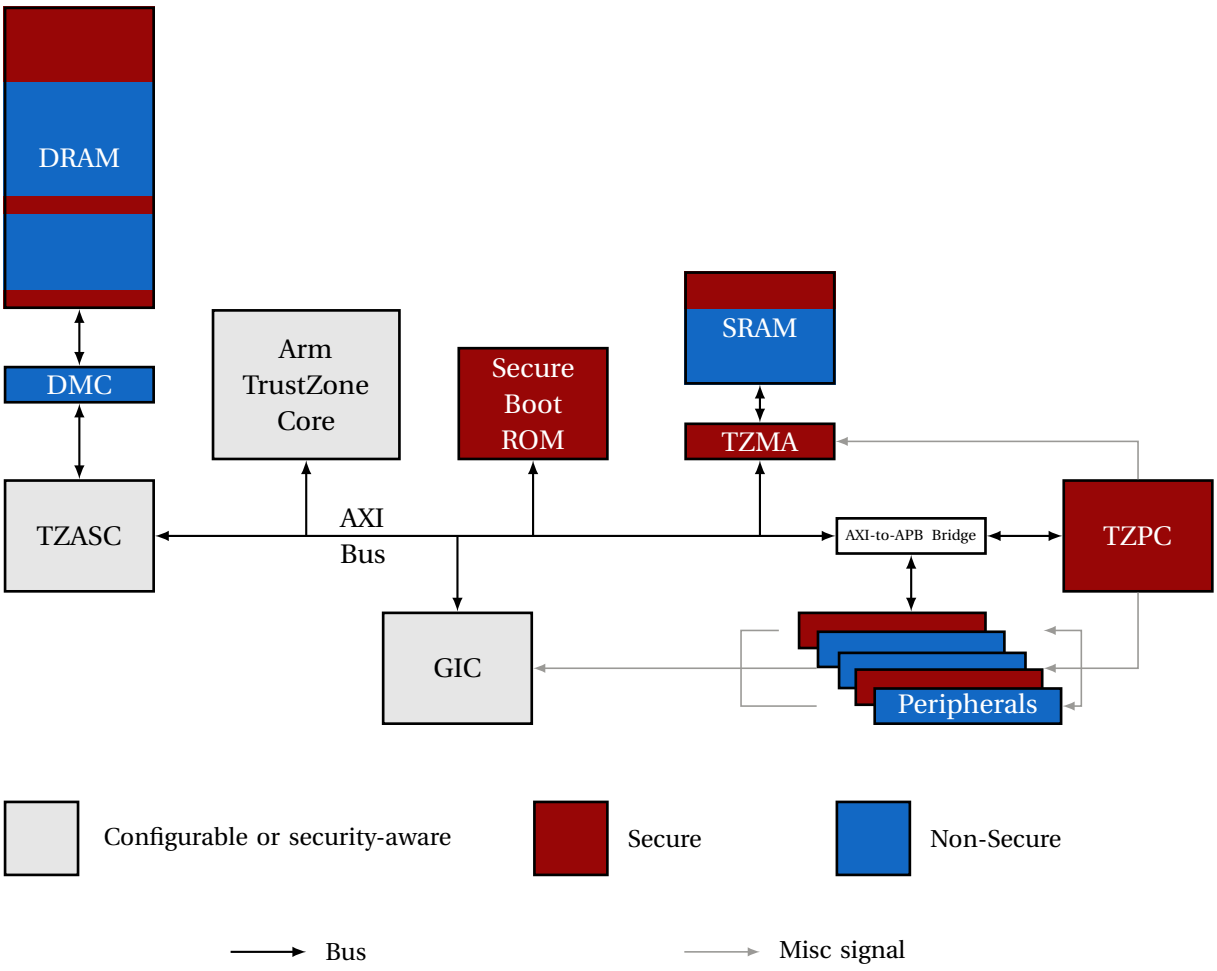


Figure 2.4: Simplified schematic architecture of the hardware components of a TrustZone-enabled system. For simplicity, only the components discussed in Section 2.2.1 are shown.

2.2.1 Enforcement

Depending on the context of execution, i.e., normal or secure, the Non-Secure (`NS`) bit is set in the Secure Configuration Register (`SCR`). The `SCR` can only be modified in a privileged execution context, typically in secure EL3 when EL3 is implemented. As depicted in Figure 2.4, numerous hardware components are then used to enforce TrustZone's security mechanisms. For on-chip static memory, e.g., the SRAM or the ROM, the TrustZone Memory Adapter (TZMA) [8] is used to split the memory into a secure region, and a non-secure (or normal) region, with the location of the separation between the two being controlled by the `R0SIZE` input signal of the TZMA. For off-chip memory, the TrustZone Address Space Controller (TZASC) [7, 10] is placed on the bus between the CPU and the Dynamic Memory Controller (DMC) – which manages accesses to the main memory. It is responsible for dynamically partitioning the main memory, and enforcing access restrictions

based on the security status of the bus transaction, resulting from the NS bit, and that of the targeted memory region. In case access is not allowed, the ASC forbids it and signals the violation to the core attempting access via an asynchronous abort. The Generic Interrupt Controller (GIC) [12] is an external interrupt controller that is responsible for the configuration, prioritization, and routing of interrupts. It supports the TrustZone security extensions and allows interrupts to be configured as secure or non-secure. Interrupts configured as secure can only be delivered to the secure world, and any attempt to modify the configuration of a secure interrupt originating from the normal world will be prevented by the GIC. Finally, the TrustZone Protection Controller (TZPC) [8] offers an interface to dynamically configure peripherals as secure or non-secure [60]. The TZPC can also be used as an interface to configure the TZMA, by connecting its output signal TZPCR0SIZE to the R0SIZE signal of the TZMA.

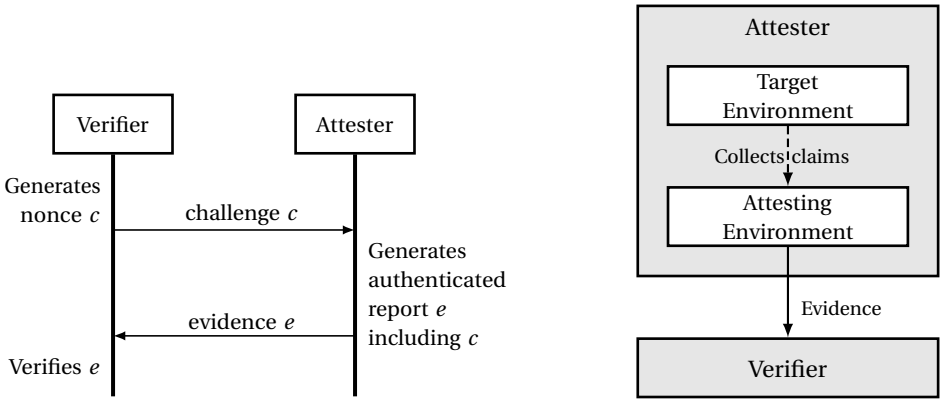
2.2.2 Limitations

TrustZone is an aging technology, it was first implemented in the ARM1176JZ(F)-S [8, 9] processor released in 2004. It is known to be vulnerable to privilege escalation attacks [24], fault injection attacks [73], as well as many other types of attacks [25]. Furthermore, its design and implementation model comes with a lot of limitations [40], such as a limited number of secure partitions (e.g., 8 for TZC-400 [7]), the large granularity of isolated partitions (32kB), the asynchronous aborts it generates on illegal accesses to the main memory, making e.g., the reconstruction of faulty instructions significantly harder than with a synchronous abort, and the fact that most TrustZone hardware components are neither standardized, nor mandatory (e.g., TZASC, and TZPC), limiting the portability of a design across different platforms. However, it is still the most commercially available trusted computing technology on Arm platforms to this day, compared to the very recent and more powerful Arm Confidential Compute Architecture (CCA) [14], which, to our knowledge, is not available on any testing board as of today.

2.3 Remote Attestation Model

A remote attestation protocol typically consists of at least two entities, each holding one or multiple roles in the attestation process, and which are often reduced to that of the Verifier and the Prover or Attester. The purpose of remote attestation is for a trusted Verifier to establish trust in an untrusted remote party, the Prover, that will need to prove it is running in a correct state in order to be trusted.

Following the glossary of the Trusted Computing Group (TCG) Attestation Framework Specification [77], and that of the Internet Engineering Task Force (IETF) Remote ATestation procedureS (RATS) Architecture [19], we refer to the Prover as Attester and elicit its structure, as shown in Figure 2.5b. The Attester and, more specifically, its Attesting Environment (AE) must perform measurements – individually called claims – of its components. Those claims are then packaged into



(a) High-level perspective.

(b) Internals of the attester.

Figure 2.5: Remote attestation process between an attester and a verifier.

an evidence that it will sign and forward to the Verifier. This signature provides authentication of the AE and guarantees the integrity of the collected claims. The measured components are collectively described as the Target Environment (TE), of which the AE can be a part.

The remote attestation protocol is triggered via a challenge-response exchange between the Verifier and the Attester, which, as shown in Figure 2.5a, is typically initiated by the Verifier. Upon reception of the challenge, the AE will collect measurements of the TE. Such measurement usually consists of a cryptographic hash of immutable data, namely the program code and program data, due to the unpredictability of the state of mutable data such as program memory, and therefore only focuses on attesting launch-time integrity, leaving runtime integrity and other security properties (e.g., availability) out of the scope of the attestation process. The measurements are then aggregated into an evidence, signed with the received challenge for freshness, and sent to the Verifier as part of the response.

A fundamental requirement in remote attestation is that the AE must be trusted to perform measurements accurately. If the AE is compromised, the evidence it produces cannot be relied upon. In order to establish trust into the Attesting Environment, it is necessary to depend on a trusted entity that will be able to attest to its integrity as part of a so-called Chain of Trust (CoT). Each link or component of this chain is verified by the previous one, with the first element being the Root of Trust (RoT).

2.3.1 Root of Trust

Although the exact definition of a Root of Trust seems to be vendor-specific [6, 41, 78, 80], some characteristics are prevalent. From these characteristics, we define the RoT as an inherently trusted, preferably small, core security component of a device that offers the necessary set of critical features

to extend or establish trust in the underlying system. It provides functionalities such as trusted or measured boot, and storage of secret cryptographic keys. It is trusted to always operate in a dependable manner, as any alteration of its behavior will go undetected. It is often implemented as a hardware chip, commonly referred to as a TPM [79], which is impractical and too expensive for smaller systems such as IoT or Edge devices. For these platforms, the Trusted Computing Group (TCG) recommends other alternatives, one of which suits our needs: the Device Identifier Composition Engine (DICE) [75].

2.3.2 Device Identifier Composition Engine

The DICE, standardized by TCG, is a capability of a Root of Trust with minimal requirements [76] which boil down to (1) secure storage of a Unique Device Secret (UDS), (2) measurement capabilities to measure the First Mutable Code (FMC), that is, the first software layer after the DICE, and (3) hashing capabilities to generate the private key for the FMC, based on its measurement and the UDS. In DICE terminology, such a private key is referred to as a Compound Device Identifier (CDI), and is used in place of the UDS for subsequent layers.

TCB Layering

DICE has a layered architecture that follows the execution states entered progressively. Starting from a base hardware layer, each layer $i+1$ is measured and CDI_{i+1} is derived from (1) that measurement, and (2) the CDI_i of the layer performing the DICE flow. The measurement of a layer with optionally other identifying information, such as configuration variables, are collectively referred to as the TCB Component Identifier (TCI). The derivation of CDIs is performed using a one-way function, typically a cryptographically secure key derivation function (KDF).

Given a key derivation function $KDF(ikm,\ salt) \mapsto okm$ with input key material ikm, salt $salt$, output key material okm, and implicit output length parameter, the CDI CDI_{i+1} of layer $i+1$ can be computed as follows:

$$CDI_{i+1} = KDF(CDI_i,\ TCI_{i+1})$$

Once the next TCB layer – here layer $i+1$ – has been constructed, CDI_i is made unavailable (i.e., securely erased or protected) and control is passed to layer $i+1$, which receives its computed CDI_{i+1} and, in turn, can execute this flow for the next boot stage, as depicted in Figure 2.6. It is important to note that CDIs can also be used as root keys for sealing or attestation purposes, with actual secret keys being derived from them for domain separation. Similar to the Program Configuration Register (PCR) in a TPM, the layering of CDIs allows the detection of modifications in previous layers, therefore binding the execution of layer i to the integrity of its TCI and that of all previous layers, creating a Chain of Trust (CoT).

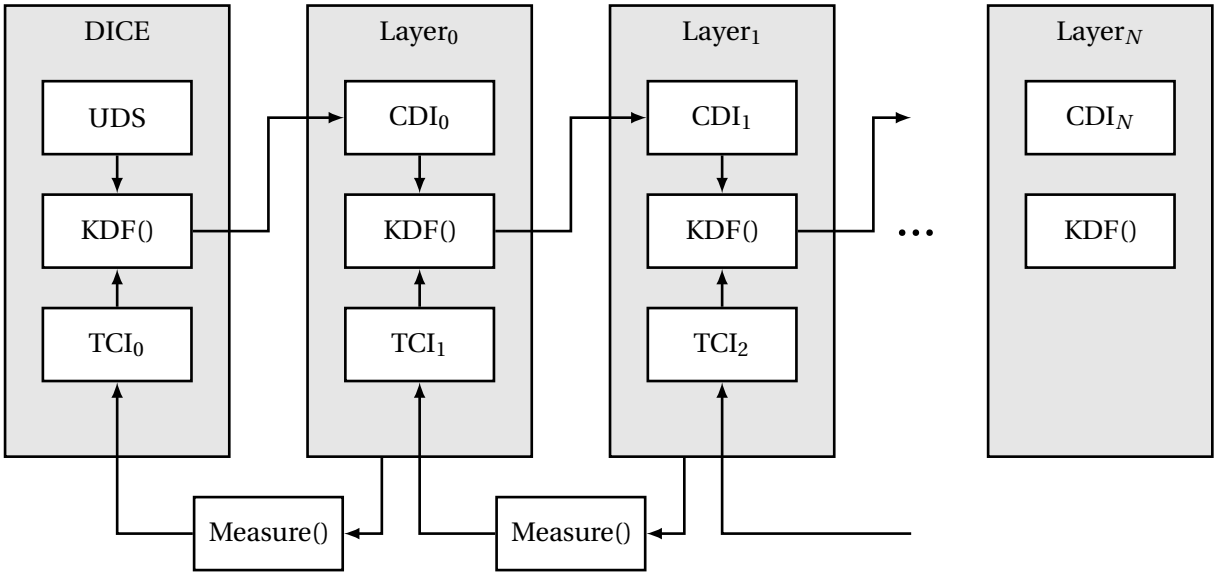


Figure 2.6: TCB layering in the DICE architecture. Each layer i measures the next one $(i+1)$ and get the result as TCI_{i+1}, which it uses as input with its own CDI_i to derive the next private CDI_{i+1} key, via the key derivation function $KDF()$. The first layer, that is, the DICE RoT, uses the Unique Device Secret (UDS) as CDI.

Chapter 3

Motivation and Goals

In this work, the aim is to provide attestable availability guarantees to tasks deemed security sensitive, specifically in the context of industrial OT systems. For that purpose, our design should be implementable on readily-available hardware, and as such should not rely on any kind of hardware modification. Consequently, our goal is also to identify the set of hardware components necessary to our design. We also highlight that, for different reasons such as Intellectual Property (IP), providers of industrial solutions often seek to keep their code and runtime data confidential from other actors present on the system.

Given the widespread adoption of TrustZone on Commodity Off-the-Shelf (COTS) Arm A-profile processors, it is currently the technology of choice for designing and implementing industrial systems on an Arm platform, while supporting certain guarantees of trust. We also investigate the use of other major Instruction Set Architectures (ISA) for the implementation of our design, such as RISC-V, and discuss our findings in Section 7.1. Due to their commercial availability, versatility, power efficiency, and general efficiency [48, 65], we conclude that, to this day, Arm platforms are better suited to the requirements of this work.

3.1 Problem Statement

Building on the above discussion, we define below the objectives for the system that we design.

G_1: Guaranteed attestable availability. Our primary objective is to provide critical tasks with guaranteed, deterministic, and timely access to the computational and non-computational resources they require. Furthermore, we want remote entities such as Supervisory Control And Data Acquisi-

tion (SCADA)[1] systems to be able to verify that the deployed real-time control loops are scheduled and executed following a predefined set of rules, which collectively compose the policies of real-time tasks.

G_2: Confidentiality and attestable integrity. Following IP and correctness requirements, code and data of critical tasks should be kept confidential from software running in the non-secure world. Simply put, the real-time tasks should be able to be loaded and run in total isolation from the untrusted subsystem. Similarly to G_1, we want a remote entity to be able to verify the integrity of the critical tasks loaded on the system.

G_3: Separation of powers. We do not wish for an omnipotent monitor that would be blindly trusted to craft policies, enforce them, and report their correct enforcement. Instead, we want applications to submit their own policy to the Monitor, while being supervised by a third party verifying that it correctly enforces the said policies.

G_4: COTS hardware. Our design shall not resort to hardware modification, but instead should build on top of existing hardware primitives available on COTS platforms.

G_5: Open, multi-tenant system. We make it a goal to design a system that supports multiple mutually distrusting tenants, each able to run their own critical and non-critical applications that can be launched dynamically at runtime, without compromising the objectives set beforehand.

G_6: Multicore usage. Safety-critical systems increasingly require sufficient compute power [66], and uniprocessor platforms may quickly reach their limits in real-time, open systems. By exploiting all available cores, the system can achieve better responsiveness, and execute comparatively more tasks at any given time.

3.2 Threat Model

We trust the hardware of the platform, and that of all external peripherals to be bug-free and work according to their specification. Once their integrity has been verified, we also trust all privileged (EL3 to EL1) software running in the secure world. We consider the established threat model in [50], in which it is assumed that any software outside the TCB is controlled by a powerful attacker. This includes all userland applications, including the real-time tasks but, as our design will show, not

[1] A SCADA is a supervisory system that can collect data from, and monitor the individual hard real-time systems in a plant.

their policy, as well as the privileged, untrusted kernel. Side-channels and physical attacks are considered outside the scope of this work. In this context, an attacker would, for instance, be able to:

- Run arbitrary computation, potentially privileged, on the system,
- Intercept and tamper with the manifest provided by an application upon loading,
- Try and launch fake critical tasks with malicious policies and maximized utilization,
- Try and lock shared resources without ever releasing them,
- Try and compromise the integrity of peripherals, e.g., by exploiting the Dynamic Voltage and Frequency Scaling (DVFS) [73, 86], causing data corruption and faulty computations, or
- Try and shutdown cores or the entire system.

3.3 Closely Related Work

For the past two decades, there has been an increasing interest in enabling Trusted Execution Environments with availability guarantees, which has led to the publication of multiple works [4, 51, 64, 84–86, 89] sharing a more or less large subset of our goals. We present here a few of these works that serve as the foundation of our study, and provide a more complete discussion in Chapter 8.

Mr-TEE [85] is a Trusted Execution Environment (TEE) that leverages TrustZone to provide safety-critical tasks with availability guarantees. It introduces a novel approach to share devices between the secure and non-secure worlds, but does not benefit from a mechanism to prove scheduling or execution of the tasks. Furthermore, its secure scheduler only runs on a single core, which, while offloading complexity from the TCB, could rapidly become a limitation in some industrial contexts where the period of critical tasks can be as low as 1ms. Aion [4] is a security architecture delivering new hardware primitives that guarantee remotely attestable progress and device access to real-time tasks, which are executed in trusted enclaves inside an open system. Aion brings hardware extensions to the Sancus [55] architecture, which is itself a hardware extension of the MSP430 architecture, contradicting our goal G_4. Hora [89] is a subsequent work providing similar security guarantees while also preventing untrusted applications from compromising the drivers of shared peripherals. It conflicts with goal G_4 due to its reliance on hardware modifications, but leverages the available multicore platform, aligning with objective G_6. RT-TEE [86] is a Real-Time Trusted Execution Environment, complementary to Aion and Hora, aiming to bring availability guarantees to real-time tasks of Cyber-Physical Systems (CPS), with careful considerations for the size and complexity of the TCB. RT-TEE targets COTS embedded platforms, which are generally closed, single-tenant systems, which conflicts with our goal G_5, and only feature a single core due to space, weight, and power (SWaP) restrictions, diverging from G_6. PEARTS [54] introduces a real-time proof of execution architecture, enabling a remote verifier to check the integrity and availability needs of

	Mr-TEE	AION	RT-TEE	Hora	PEARTS	Tyche	Our work
Guaranteed Attestable Availability	◐	●	◐	◐	◐	○	●
Confidentiality and Attestable Integrity	●	●	●	●	◐	●	●
Separation of Powers	○	○	○	○	◐	●	●
COTS Hardware	●	○	●	○	●	●	●
Open, Multi-Tenant System	●	●	○	○	○	●	●
Multiprocessor Usage	○	○	○	●	○	●	●

Table 3.1: Comparison of our envisioned design with earlier works, with regard to our goals. ● means a goal is part of the design, ○ means it is not, and ◐ means a goal is only partially implemented. For instance, ◐ for G_1 here either means that availability guarantees are provided, but are not verifiable, or that availability can be attested but is not guaranteed through trusted scheduling.

real-time applications executing in the normal world. It is aimed at closed, low-end, single-core IoT devices, and leverages the TrustZone architecture to host its attesting environment. It runs real-time tasks in the normal world, and does not provide any guarantees of confidentiality. The idea behind objective G_3 comes from the work of C. Castes, et al. [22] in which they reflect on the monopoly of powers exercised by trusted software, and propose an alternative architecture within which the so-called powers of (1) emitting policies, (2) enforcing policies, and (3) attestation are all split across different components of the system. To summarize, Table 3.1 displays an overview of how the design of the closely related works mentioned above compares to the goals of this work.

Chapter 4

Design

In this section, we discuss the design choices of our system. We first elaborate on the elements required to bootstrap it, and follow with the description of its components, which we organize into three main categories.

4.1 System Requirements

Based on the requirements for DICE implementation (see Section 2.3.2) and the technologies offered by the TrustZone security extensions (see Section 2.2.1), we elicit the minimal set of hardware requirements to provide our system with the usability and security guarantees that we derived in Section 3.1.

4.1.1 Hardware Components

To implement DICE, a secure mechanism for storing the Unique Device Secret (UDS) is required, as specified in [38]. To ensure availability, TrustZone hardware components, such as the Protection Controller, Address Space Controller, and a TrustZone-aware GIC, are leveraged to enforce isolation. These components allow us to restrict peripheral and interrupt access to the secure world, assign the memory of critical tasks to the secure world, and enforce access control policies on system bus transactions.

In order to bootstrap the availability of compute resources, we need a mechanism to dependably retrieve control over the system, so that new scheduling decisions can be taken when required. This necessitates a timer device, immutable from the non-secure subsystem, which triggers at configured intervals of time and traps to the secure environment. This type of timer can be available

in two ways on Arm platforms: (1) as a processor timer, that is, the EL1 secure physical timer (`CNTPS_EL1`) [13], or (2) as an external SoC timer configurable as a secure device, that is, in the presence of a TrustZone-enabled GIC and memory controller.

We survey about 30 Cortex A-profile multicore boards in order to identify evaluation kits that suit our needs, and report our findings in Appendix A for 19 of them, for which we could find sufficient relevant documentation. Among those 19 boards, we find that 13 of them have the necessary requirements.

4.2 The Three Powers

In order to decouple the power to put law into action, make the law, and judge the law, we split our design into three main entities, following the political model of separation of powers [26]. Similar to [22], we make an analogy between these three powers and the three concepts of enforcement, policy enactment, and attestation, respectively. Not only does this separation allow users to avoid having to blindly trust a single omnipotent entity, it also allows one to reason more clearly about our design and how to provide and verify availability guarantees that comply with the requirements provided by users. We show an overview of the architecture of our design in Figure 4.1, and discuss its different components in the following sections.

4.2.1 Executive: the Availability Monitor

In order to provide availability guarantees to real-time tasks, we must ensure the enforcement of the policies of those tasks. To fulfill that function, we propose an availability monitor (AM), whose role is to schedule critical tasks according to their attached policy, and provide them access to required computational and peripheral resources in a timely manner. The AM is designed as a composite entity, with components working hand in hand to provide the required properties. Similar to [86], we design a two-layer hierarchical scheduler [31] which comprises a world scheduler at the top layer, and sub-schedulers for each world at the bottom layer: the secure scheduler and the normal-world scheduler. Although the world and secure scheduler can be thought of as the head of the AM, they are not omnipotent as they mainly focus on sharing computational resources, and delegate management of external resources to Device Agents (DA) [1]. The DAs, commanded by the secure scheduler, focus on enforcing temporal and spatial isolation of shared resources to ensure their availability to critical tasks.

[1] In multiprocessor resource sharing protocols, particularly in the context of the Distributed Priority Ceiling Protocol (D-PCP) [63], Resource Agents was the name given to special tasks that handled access to resources on behalf of requesting tasks. We reuse the name of agent in that context.

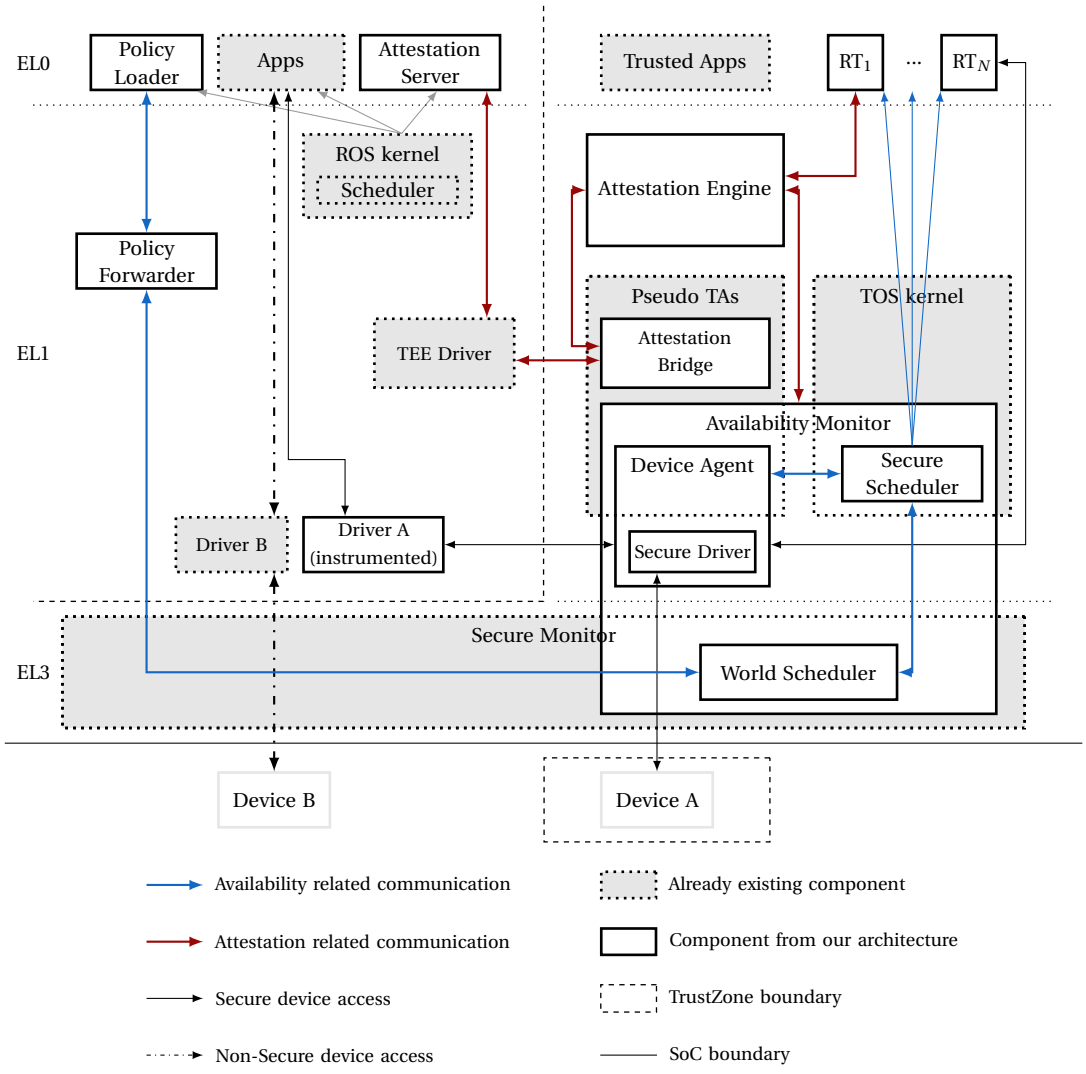


Figure 4.1: Overview of the architecture of our design. For simplicity, communication of the normal world scheduling trusted applications (TAs) have been omitted. Note that all communication between the normal and secure world has to go through the secure monitor in EL3; for simplicity, we omitted that additional hop.

The World Scheduler

The world scheduler is a preemptive scheduler that will preempt execution of the current world to resume one or the other, based on the requirements elicited in policies. It is responsible for enforcing budget constraints on the sub-schedulers, and is agnostic to the actual scheduling algorithm they implement. It executes at the highest level of privileges on the platform, and as such, its set of functionalities should be kept as small as possible to reduce its attack surface and allow for formal verification. In an industrial context, it is crucial that critical tasks – which need to execute in the secure world to benefit from the guarantees our design provides – complete before reaching their deadline, even at the risk of starving the normal world. For that reason, we give the secure scheduler an infinite budget and only retrieve control when it encounters idle time. Given this constraint, we consider the NS scheduler as an aperiodic task and envision a background scheduling algorithm to handle such tasks (see Section 2.1.1), which means that the NS scheduler will only be scheduled when its secure counterpart is not busy. This method presents the advantage of having no impact on the schedulability of critical tasks, at the risk of starving the normal world. Background scheduling also makes it easier to reason about the schedulability of critical tasks, while facilitating the implementation of the world scheduler, therefore reducing the complexity of the TCB.

The Secure Scheduler

The secure scheduler is the section of the AM that is responsible for enforcing availability guarantees of computational resources to critical tasks. It leverages a partitioned, priority-based, event-driven, preemptive, fixed-priority scheduling design. Since safety-critical systems increasingly depend on multicore for sufficient compute power [66], we design our scheduler to take full advantage of all application processors on the platform, but take a partitioned scheduling approach. This means that each task remains assigned to the same core for its entire lifetime; this assignment is sometimes referred to as the affinity of the task. Compared to global scheduling, partitioned scheduling has the advantage of being less complicated to implement, helping towards our goal of keeping a small TCB, as it does not require complex load-balancing algorithms, and many techniques used in uniprocessor scheduling can be applied or easily adapted to it. Moreover, it does not suffer from the performance overhead of migrating tasks across cores [32], which can increase the scheduling latency. Until a first task gets scheduled on any given core, the secure scheduler responsible for that core acts as a polling server, periodically checking if it has received a critical task from another core, and suspending itself until the next polling tick if it has not. When a first task is received, the polling mechanism in which ticks occur at regular intervals switches to an event-based approach, in which the scheduler is configured to tick on the next event, which has a relative upper bound equal to the execution time of the job, defined in the provisioned task policy. In this case, the next event refers to the suspension of the current task leading to the scheduler retrieving control over the core, which can occur either because the task has completed, or because of its preemption by a higher priority task, or because it has blocked on a shared resource. At every tick, the scheduler takes a

new scheduling decision, which consists in (1) retrieving the highest priority task in its pool of ready tasks, and (2) retrieving the next nearest release of a job. The next task is then scheduled on the core, and the next scheduling event is configured to be the earlier between the end of the execution of the current task, and the next release. The configuration of the next tick allows the scheduler to impose an upper bound time limit after which it will retrieve control over the core, so that it can take a new scheduling decision. This mechanism is realized through the use of a core-private secure timer. This timer is configured such that it can only be programmed from, and can generate interrupts routed only to, a privileged secure context (i.e., S-EL1 or higher). Furthermore, its interrupts are assigned the highest possible priority on the platform, above all others, thereby taking precedence over any concurrent interrupts. In the case where no task is ready to be scheduled on the core, we say that the scheduler becomes idle. When the scheduler encounters idle time, it is able to suspend itself and gets preempted by the world scheduler.

To share or not to share. Depending on the total utilization of a core by the set of critical tasks it has been assigned, as well as the requirements of the end-customer, it might either be interesting or too demanding to share the core between the secure and non-secure worlds. If the secure scheduler does not release the core while idle, it puts the core into a low-power state until it is awakened by the next scheduler tick. On the other hand, when the core is shared between the two worlds and returned by the trusted scheduler, a context switch will occur when the secure scheduler becomes idle. This switch, operated by the world scheduler, comes at a certain cost that we measure in Section 6.1. While the penalty incurred by such an operation is acceptable when returning control to the normal world, as it does not host applications with a hard deadline, this performance overhead might become problematic when a critical task needs to be scheduled. Especially in the cumbersome case where the next secure scheduling event would happen during the transition from the secure world to the non-secure world. For that reason, we leave the option to the system designer to choose whether or not to allow sharing of a core between the two worlds.

The Device Agents

The secure scheduler interacts with entities that we refer to as Device Agents (DA), which are responsible for enforcing strict spatial and temporal isolation on the peripheral they manage, collectively ensuring the availability of shared resources on the platform. DAs act as wrappers around secure drivers, and provide a firewall-protected interface to the configuration and functionality of the device they protect by monitoring every access.

Spatial isolation. Spatial isolation is a necessary component for bootstrapping availability on the system, as it enforces strict access control to preserve the device logical and physical integrity. When initializing, DAs leverage the security extensions available on the platform to isolate their device

from the untrusted NS world, and enforce access control at the bus level. Generally, a program can communicate with a peripheral in three main ways: (1) via Memory-Mapped IO (MMIO), that is, the device memory is mapped to the address space of the system by configuration of the page tables in the memory management unit (MMU), (2) via interrupts, generated by the peripheral and routed to a core, where interrupts are enabled by configuring the GIC and the peripheral through MMIO, and (3) via Direct Memory Access (DMA), where the device bypasses the CPU and directly reads from or writes to the main memory, at addresses configured by the driver through MMIO. Mapping a device to the secure world thus consists in isolating its memory-mapped registers, isolating the interrupts it generates, and in the case of DMA devices, configuring the device as secure so that it is granted read and write access to secure memory. Once secure devices are properly isolated, the DAs are able to monitor every MMIO access coming from the normal world and secure applications, as it is the only available interface at their disposal. The availability of a peripheral can be disrupted by supplying malicious values to control registers, targeting two main attack vectors [86]: either its logical integrity (e.g., changing the detection thresholds of a proximity sensor) or physical integrity (e.g., by over-raising the frequency of a core [73]), causing the peripheral to act unexpectedly. A Device Agent can prevent such behaviors by first filtering requests, that is, by allowing or disallowing based on the permissions of the requester; and second by sanitizing the content of the requests, e.g., by verifying the configuration values are within the operational bounds of the device. Finally, a remaining trivial attack that a fully compromised normal world OS could perform would be to simply shut down the system. In order to prevent such an attack, as well as more elaborate ones targeting, for instance, the Device Voltage and Frequency Scaling (DVFS) [73, 86], we can firewall the power management functionality of the SoC behind a dedicated Device Agent.

Temporal isolation. In general, a real-time system comprises multiple tasks, all of which may require access to different resources. In an ideal world, every task requires a different set of peripherals, which would make scheduling of resources straightforward. However, the reality is quite different, and tasks running in parallel (e.g., with intertwined scheduling of jobs on a single core, or at the same time on different cores) can compete for the same devices, leading to contention issues. To mitigate these issues and handle the sharing of resources on the system, we use a suspension-based Multiprocessor Priority Ceiling Protocol (MPCP) (see Section 2.1.2), and consider all SoC-external resources to be global resources; that is, all external resources can be acquired by tasks running on any core, provided the required resources are declared in their policy. The MPCP protocol imposes an upper bound limit on priority inversions, equal to the access time of the critical section of the lower priority task [27, 62]. Device Agents further constrain this bound to the provisioned execution time of the owning task, as specified in its task policy, after which it is forced to return the device to the Availability Monitor. It is therefore the responsibility of the policy provider to ensure that the parameters specified for a task T_1, potentially blocking a higher priority task T_2, are carefully chosen so that the blocking time induced on task T_2 does not prevent it from completing before its deadline.

When a change of ownership is observed, the DAs ensure their peripheral is reset to a neutral

state. This mechanism prevents (1) the leakage of potentially private information to the normal world or between mutually distrusting real-time tasks, and (2) corrupted behavior resulting from a critical task that expects the peripheral to be in a given state while it is not.

Peripheral sharing across worlds. At design time, depending on the requirements of critical-tasks, a set of peripherals can be mapped to the secure environment. In order to provide availability guarantees to real-time tasks by applying temporal and spatial isolation of required devices, it is mandatory that all peripherals that may be claimed by critical tasks are assigned to the secure world. From this point on, each individual device can either be shared across both worlds, or exclusively owned by the secure world. A device is exclusively owned by the secure world if and only if the normal world has no way of accessing it, no matter the level of privilege (EL2, EL1, or EL0) it is executing at. This access control policy is enforced by TrustZone and configured by the AM. While exclusively secure devices present advantages in terms of simplicity of design and security, as no interaction is possible from outside of the TCB; it prevents normal world applications from legitimately using them. In order to share devices, we use a split-driver model [64, 84–86] in which the drivers of such peripherals present a normal and a secure split. The normal split exposes the feature-rich application-facing part of the driver, while the secure split exposes the low-level peripheral-facing part, directly communicating with devices. When a normal world application wants to use a peripheral, it calls an interface exposed by the NS driver, which will need to invoke the DA of that peripheral in order to interact with it.

Interrupt management. An important aspect to consider when handling real-time tasks with strict timing requirements is the case of interrupts. In order to perform a Denial of Service attack on the Availability Monitor, a malicious untrusted OS could configure a non-secure peripheral such that it continuously triggers interrupts. The interrupts handled by the trusted OS and those that are not are treated by the GIC as two different kinds of interrupt [82]. When executing, our TEE masks the interrupts that it did not configure, and unmasks them when returning control to the normal world, thereby mitigating such an attack vector. Although non-secure interrupts are handled by our design since they are outside the TCB and constitute an important attack vector, we allow greater flexibility in how secure interrupts are implemented. Since (1) the set of secure peripherals (and therefore their driver with DA) is chosen at design time, and (2) the access control policy and actual implementation of specific Device Agents depend on the requirements of the end-customer, we let the system designer carefully decide how interrupts are handled and shared in the secure world. Furthermore, our design currently does not support sharing interrupts from secure devices to normal world observers. We refer to the state-of-the-art [85] for such a mechanism. In order to share secure interrupts to the normal world, [85] creates a communication channel between the S and NS drivers, which they refer to as *Shared Secure Peripherals Notifier* (SSP Notifier) which has a normal world Notifier on one end, and a secure world Notifier on the other end. When a secure device raises an interrupt, it is handled by the S driver, which in turn triggers a normal world interrupt via the

secure world Notifier, to which the normal world Notifier is subscribed, and can then forward to the normal world driver for further handling.

4.2.2 Legislative: Real-Time Policies

In the previous section, we discussed how the Availability Monitor can enforce the availability of computational and non-computational resources to real-time tasks with a hard deadline, following strict requirements elicited in the so-called task policies. We now discuss these real-time tasks with their task policies, and how to securely provision them to the AM. Note that, except for availability, secure aperiodic tasks also benefit from all the security guarantees discussed below.

Secure Applications

Following the goals elicited in Section 3.1, the requirements for critical applications boil down to ensuring and attesting their integrity, preserving their confidentiality, and providing them with availability guarantees. As discussed previously, once loaded with their attached policy, availability is enforced by the Availability Monitor. With regard to secure applications (not their policy), this leaves open three questions: (1) How to provide confidentiality at rest and in use? (2) How to ensure actual loading? and (3) How to ensure and prove their integrity?

Confidentiality. Confidentiality in use is warranted by the TrustZone part of our system. Once loaded in the Trusted Execution Environment (TEE), tasks execute in complete isolation from any software in the normal world, regardless of its level of privilege. Moreover, since secure tasks distrust each other, we leverage the MMU to configure userland page tables and isolate their respective address space, ensuring the confidentiality of their program code, data, and memory. Regarding confidentiality at rest, confidentiality of task binaries is achieved through the use of cryptography. Our TEE should support the loading of encrypted binaries, which it shall decrypt inside the secure world so that no untrusted actor on the system can access the IP code.

Guaranteed loading. Generally, the binaries of secure applications are stored on the disk and retrieved by the Trusted OS (TOS) when loading them [82]. Since porting device drivers to the secure world increases the size and complexity of the TCB, TOSes generally ask their counterpart feature-rich OS to perform actions on their behalf. Examples of such mechanisms are reading from, and writing to the disk. In order to perform disk operations, a TOS issues a request in the form of a Remote Procedure Call (RPC) to the untrusted OS that implements a file system as well as necessary disk drivers. The normal world OS then performs the read or write operation and returns from the RPC. When we consider all untrusted software to be under the control of an attacker (see Section 3.2), this mechanism can break all availability guarantees (e.g., CVE-2025-46733). For

instance, a malicious kernel could simply deny requests to the file system, preventing the TOS from loading secure applications. In order to remediate that issue, our design should support the pre-loading of critical tasks. Pre-loaded tasks are appended to the TOS image and are thus loaded at the same time as the trusted kernel. They are then stored in secure, read-only memory and can be loaded without the intervention of the non-secure OS.

Integrity. The integrity of trusted tasks can first be verified in two ways, depending on whether the task was pre-loaded or fetched from the disk. In the former case, the task binary is part of the trusted kernel image, whose integrity is checked by the Root of Trust at boot time, prior to being run. As the binary is then stored in trusted read-only memory, its integrity is preserved and does not need to be asserted again when mapped into memory. When the trusted task is fetched from the disk, we are again faced with two different cases, depending on whether or not the end-user decided to encrypt the binary for confidentiality at rest. Should it be the case, our TEE shall support authenticated encryption schemes to securely store binaries, such as AES in GCM mode [33]. Authenticated Encryption is an encryption scheme that provides confidentiality while ensuring integrity of the data. Finally, if the binary is not protected for confidentiality, but still requires authentication, its policy should include a field specifying the checksum of the task binary.

Task Policy

Task policies are the set of properties and requirements that define critical real-time tasks. They specify the parameters of real-time tasks as defined in Section 2.1.1, which are their period, execution time, core affinity, and set of required shared resources, as well as a unique identifier binding them to a unique task (i.e., binary of a Trusted Application), and an explicit priority field. An example of such a policy is displayed in Listing 4.1. The task policies are user-provided with the help of an unprivileged *policy loader* in the normal world (i.e., NS-EL0). The *policy loader* then sends a request to the privileged *policy forwarder*, which forwards the policy to the world scheduler, subsequently passing it onto the secure scheduler. This exchange is depicted in Figure 4.1. Collectively, task policies form the scheduling policy of the system, making them a central part of the availability guarantees and a potential attack vector. For this reason, they should be authenticated. Authentication is used to verify that the policy has not been tampered with, and that it was defined by an authorized entity.

Integrity protection. A malicious entity in the non-secure world could try to modify the properties defined in a task policy that is being forwarded to the secure world. We add a mandatory `key` field in polices that holds a cryptographic hash of the values defined in that policy. The cryptographic scheme used to hash the policy must provide preimage resistance, second preimage resistance, and collision resistance, so that it is not possible for an attacker to alter the policy without altering the computed hash too.

Authorized policies. By simply hashing the policy, it would be trivial for an attacker to craft a malicious policy, or tamper with an existing one, and compute a new valid hash. We use a cryptographically secure signature scheme to circumvent this issue. In public key cryptography, a signature scheme allows an entity to sign a payload with a key only known to them; this key is referred to as the secret key. The signature is then sent alongside the payload, and can be verified by a third party with the public key of the signer. With this mechanism, it is possible for our TEE to verify not only the integrity of a policy, but also to ensure that it was created by an authorized entity. The storage of the secret key and the signing of the policies should be done in a secure environment and never on the system, as a malicious entity could record the secret key being used and later craft their own policies.

Replaying the policies. A malicious entity could record a valid policy and try to replay it multiple times in order to flood the secure scheduler and disrupt the system. Our design handles that case and does not authorize more than one instance of the same critical task at a time. Another attack could consist of an entity recording a maximum amount of valid policies to attain the same result as the last attack. We argue that it is the responsibility of the end-user to ensure that only the critical tasks required by a system exist on the said system, restricting the number of scheduled real-time tasks to the intended count.

Rollback protection. Currently, our design does not support rollback protection for the policies. However, we argue that such an attack vector can be detected by a remote verifier leveraging our Attestation Engine.

Denied forwarding. Finally, a corrupted *policy forwarder* could simply drop the policies instead of forwarding them. To mitigate that issue, our TEE should be able to support pre-loaded scheduling policies, similarly to the pre-loaded secure applications.

```
uuid        = 0156eabe-ec72-4850-9e95-a55174961383
period      = 5000
exec-time   = 1000
priority    = 10
affinity    = 1
// considered empty if omitted
peripherals = 36, 49
// optional field
checksum    = 7db9cfa04aed2cf8fb3ebc6285061e9d14b1df31d2d4fa241d39aec0be5ce24b
key         = 00828fe43345a3816cce0a11cf75ae4bb4f517c90a9882ed3e17b4bb959dacfd
```

Listing 4.1: Example of a task policy.

4.2.3 Judiciary: the Attestation Engine

In this section, we discuss the design of our Attestation Engine (AE) for availability guarantees. Its purpose lies in ensuring that the Availability Monitor performs its duty in accordance with the received policies. In other words, the AE must be able to verify and show to a remote party a proof of unaltered execution for a target real-time task. [54] introduces and formalizes the requirements for this new security goal, which they refer to as Real-Time Proof of eXecution (RT-PoX).

The Trust Anchor

We use a Device Identifier Composition Engine (DICE) Root of Trust (RoT) to act as the trust anchor of our Attestation Engine. We leverage the layering architecture of DICE to extend the Chain of Trust up to the TOS, and generate two private keys (so-called CDIs) that are bound to both the hardware and software state of the device. One advantage of such a design is that, apart from a unique hardware secret, all private keys are generated at runtime and do not need to be stored on the disk. Among the two previously generated keys, the sealing CDI is used for sealing purposes and will serve as a root key for (1) decrypting the binaries of encrypted critical tasks, and (2) encryption and decryption for secure storage functionality in TAs. The other key, called the attestation CDI, will be used as a seed to derive a pair of cryptographic keys that will later be used by the Attestation Environment (AE) to sign evidence.

DICE in more detail. Starting from a hardware-stored unique device key, called in DICE [75] terms a Unique Device Secret (UDS), our design derives a new secret key (CDI) for each booting layer. Each layer constructs the key for the next one and destroys its own CDI before passing control forward. As described in Section 2.3.2, a layer derives the CDI for the next booting stage from its own CDI and a measurement of that stage, along with its configuration parameters, which together form the TCB Component Identifier (TCI). This layering architecture presents the advantage of producing keys that are not only bound to a unique device, but also to the state of the software running on the platform. Any incorrect state in the TOS or prior boot stages will produce different keys, ultimately resulting in an inconsistent attestation signature that can be detected by a remote verifier. As depicted in Figure 4.2, this Chain of Trust extends in the secure world up to the TOS, which can now be relied upon to execute attestation.

Key rotation in the TOS. In our design, the TOS needs to use private keys for multiple purposes, such as TA decryption, secure storage, and attestation. As private keys should only be used for a single purpose and a single cryptographic algorithm, we use a Key Derivation Function (KDF) to generate new keys from the CDI and obtain domain separation. However, as the TOS might need to probe the CDIs multiple times, we cannot simply erase them from memory once they have been

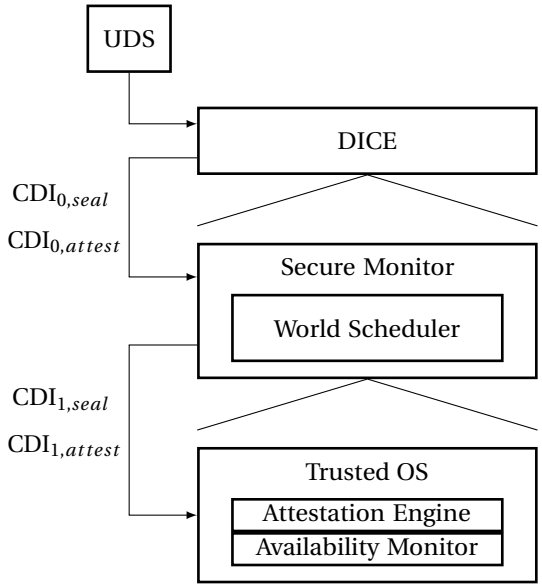


Figure 4.2: Illustration of the DICE layering in the secure environment of our architecture. Each layer measures the next one and produces CDI keys based on this measurement, its own CDIs, and optionally configuration parameters of the next layer.

first consumed, as is the case with the previous layers. As the KDF offer forward secrecy guarantees, a leaked private key derived from the CDI cannot be used to retrieve the original CDI. However, the converse is not true, and a leaked CDI would allow an attacker to retrieve all private keys derived from that CDI. In order to mitigate this issue and constrain the set of leaked keys to only the leaked CDI and the next derived keys, we use a mechanism similar to the double ratchet algorithm [57]. To put it simply, every time a CDI key is probed, it is replaced by another key derived from it (one root key ratchet), leveraging a different KDF than the one used by the other components of the TOS that derive their own private keys from it (ratchet on the derived keys). We illustrate this process in Figure 4.3. Although this mechanism allows our system to constrain the set of leaked keys, the top CDIs should still be discarded in order to regenerate new keys whenever a private key is leaked. Due to the design choices, regenerating new CDIs simply consists of updating the TOS.

Attesting the Availability Guarantees

Now that the foundation has been established for our Attestation Engine to reliably fulfill its function, we need to define the requirements to provide proof of unaltered execution.

Properties for RT-PoX. [54] enumerates five properties that must be maintained to attest to RT-PoX for a target task T on the system. Property (1) stipulates that the binary of T must be immutable. Our design achieves this by loading T in the hardware-protected secure world, and by configuring

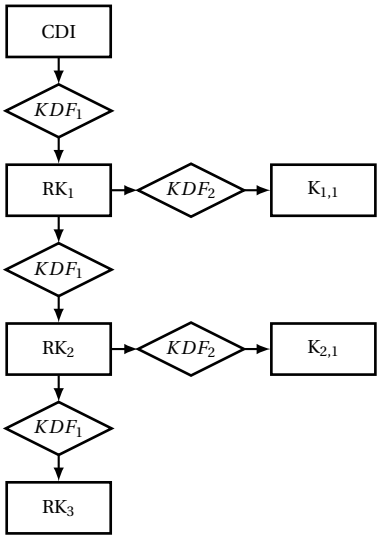


Figure 4.3: Schematic representation of the key rotation mechanism for secret keys derived from a CDI.

the pages where the binary is loaded as not writable. The page tables dedicated to the Trusted Applications can only be configured by secure privileged code (i.e., at least S-EL1), which is trusted not to be malicious. Property (2) requires that the integrity of T's control flow must remain unaffected by code transfers, such as those generated by interrupts, system calls, or the parallel execution of untrusted software running on the platform (recall that to T, untrusted software is all code running in the normal world as well as other secure applications). Moreover, the jobs of T shall exclusively start and terminate at their valid entry and exit points. The design of the secure scheduler and Device Agents, as described in Section 4.2.1, ensure this behavior. Property (3) specifies that all data used by T must not be tainted by untrusted software. In our system, the only memory addresses that T shares with untrusted software are those of shared peripherals. When a device is passed from one application to another, i.e., when a change in the owner of a device is observed, the Device Agents restore the state of their device to a clean state, thereby acting as a sanitizer for the tainted values, ensuring that property (3) holds. Next, we lay down the requirements for properties (4) and (5), and discuss how the AE is designed to hold these properties in the next paragraphs. Property (4) mandates that the evidence generated by the attestation environment includes information about any delays (i.e., missed deadlines) perceived by T. Finally, property (5) prescribes that the Attestation Engine shall not interfere with our Availability Monitor.

The Attestation Engine. Compared to the time constraints under which some real-time tasks can be required to run, the attestation process is quite costly (see Chapter 6 for details about those measurements). Given how critical, periodic tasks are prioritized over aperiodic ones in our current design (recall the background scheduling discussed in Section 4.2.1), the Attestation Engine could suffer from starvation if scheduled on a core where the total utilization of critical tasks is too high.

Depending on the requirements of the end-user, we propose two solutions to this issue. The first option is to let users take advantage of their multicore platform to schedule the AE on a core that is not heavily loaded. Alternatively, the attestation process may be scheduled as an aperiodic task with an assigned priority, whose exact value depends on the requirements of the other tasks running on the core. Regardless of the chosen option, the AM will always execute with higher priority than the AE, and will always be able to preempt it, thereby ensuring that property (5) is maintained.

We now discuss the actual attestation process performed by the Attestation Engine. It is first important to note that the AE requires the secure scheduler to record every occurrence of a missed deadline for each task, as it is a necessary condition for property (4). The attestation starts when a remote verifier queries the AE by sending a challenge accompanied by the unique identifier of a target task. It then reads the AM memory and produces a report of that task. The set of information contained in the report uniquely identifies the task and is identical across all correct reports for that task, thus representing a fingerprint for the task. This fingerprint holds information such as the identifier of the task, the content of its policy, the number of missed deadlines, its current state (e.g., runnable, done, dead, etc.), whether it is allocated a valid thread, and whether it was recently run. Then, the AE reads the address space of the target task and measures its different read-only sections. The fingerprint and the program read-only code and data are then cryptographically signed with the engine's private key (derived from the attestation CDI), as well as the challenge for freshness. This constitutes the evidence sent to the remote verifier. Although TrustZone and the Availability Monitor preserve the integrity of the loaded binary throughout its lifetime, integrating it into the attestation evidence allows a remote verifier to ensure that the correct version of the critical task is installed on the system. Given that a valid task fingerprint, that is, one that shows proof of execution for all the jobs of a task, will always yield the same hash, it is possible for a remote verifier to detect whether a task experienced delays in execution. Therefore, satisfying property (4).

Chapter 5

Implementation

For functional validation of our design, we implement a prototype on the TI TMDS64 evaluation module that is equipped with an AM6442 SoC bringing two Cortex-A53 application cores running at up to 1.0GHz, and a Cortex-M4F running at up to 400MHz. This platform has all the hardware components that we require (recall Section 4.1.1), as observed during our survey with the results detailed in Appendix A. The SoC also comes with four Cortex-R5F cores, which the current design does not use. The M4F core serves as the hardware Root of Trust and implements a closed-source security firmware for key and security management. As the platform does not provide access to the First Stage Boot Loader (FSBL) in the ROM code, we implement the first DICE stage in the second stage of the initial boot sequence, known in Arm terms as BL2. For our implementation of the Device Identifier Composition Engine, we implement the DICE profile specified in [38], as it suits our needs and fills in the gaps left intentionally in the TCG DICE specification [75] to offer flexibility in how it is implemented. To do that, we extend the Secondary Program Loader (SPL) of U-Boot 2025.01 [30], as U-Boot is the reference boot loader for our evaluation kit. We use Arm Trusted Firmware-A v2.12.0 [81] as the secure monitor in EL3, responsible for the context switches between the normal world and the TrustZone, since it is a reference open-source implementation for Arm platforms. We then augment it with our world scheduler, as well as a DICE runtime service used by the secure and non-secure kernels to retrieve their CDIs. To realize our Availability Monitor and Attestation Engine, we expand OP-TEE OS v4.5 [82], a GlobalPlatform TEE [36]-compliant TEE that initially does not implement a scheduler. Although there exist other TEEs [5, 61, 83], OP-TEE is open-source, thoroughly documented, and has been extensively used in research, which makes it the ideal candidate for our needs. It runs alongside Linux v6.12.17 [35] in the non-secure world, which also conforms to the GlobalPlatform TEE standards, providing a standardized communication interface between the two kernels.

The schedulers' timer. The evaluation platform that we use provides both secure processor timers and multiple board timers (i.e., devices external to the SoC). On Arm platforms, devices can generate

two types of interrupts, depending on how they are connected to the SoC. An on-SoC device will generate Peripheral Private Interrupts (PPI), where an external device will generate Shared Peripheral Interrupts (SPI) [12]. A PPI is routed directly through the redistributor of the GIC, which, in simple terms, means that the interrupt is routed directly to the processor on which the interrupt was generated. On the other hand, SPIs will first be routed to the distributor of the GIC, where a routing decision will be made to target a specific processor, based on the configured affinity of the interrupt. Although acceptable ([85] measures the latency of the external EPIT timer on their evaluation platform), we argue that the increased latency resulting from the use of an external timer can easily be avoided by using the secure physical core timer. Given how tightly coupled the world scheduler and secure scheduler are, and the fact that there is only a single secure timer available per core, the same timer is shared for both schedulers. This is achieved by dynamically (1) reconfiguring the timer interrupt as EL3 or EL1 via the GIC, and (2) enabling the `ST` bit in the Security Configuration Register (`SCR_EL3`) to trap EL1 accesses to the secure timer while the world scheduler has claimed ownership.

The secure scheduler. We build our secure scheduler on top of the existing thread management and TA management in OP-TEE. First, this allows us to increase the TCB only with the scheduling logic, and implementation details regarding the integration of the scheduler within the existing infrastructure. Second, although not yet integrated into the current prototype, this would allow the secure scheduler to handle Trusted Applications scheduled by the normal world kernel, which can still coexist with the existing implementation but without the availability guarantees provided to real-time tasks. When TAs are scheduled, whether by the secure scheduler or the NS OS, a session is initialized, stored in an internal data structure, and a handle for the session is returned to the caller for further management (e.g., invoke a command, close the session). To prevent the NS world from interfering with the sessions managed by the trusted scheduler, we divide this data structure into a secure part and a non-secure part, with the secure portion being inaccessible to the untrusted OS. Moreover, since TA management relies on synchronization mechanisms (e.g., spinlock, mutex) to maintain coherency across cores, we use separate locks for the secure and non-secure sessions. This prevents the normal world from delaying the secure world when managing TAs.

Critical real-time tasks. The real-time tasks scheduled by the secure scheduler are implemented as OP-TEE Trusted Applications. In Appendix B, we show how one can very simply turn a control loop written as a classic C program into a control loop intended to be scheduled by the secure scheduler. Standard TA properties are specified in the GlobalPlatform TEE specification. Among the required properties are the flags of the TA that specify its behavior (e.g., single instance session, concurrent sessions, keep alive). To provide an additional layer of protection for critical tasks, we introduce a new flag, namely `TA_FLAG_SCHEDULER_PROTECTION`. When defined, this flag protects the main entry point of the Trusted Application (i.e., `command = 0`), ensuring that it can only be invoked by the secure scheduler, and not by the untrusted OS or other TAs on the platform. This is

useful when implementing, for instance, IO images. An IO image is scheduled as a critical real-time task, periodically probes a sensor, and stores the measurements in a buffer accessible by other periodic or aperiodic applications. In such a case, the main entry point running the logic described above should only be invoked by the secure scheduler, and another entry point should be accessible to other applications to read measurements from the buffer. Finally, to mark the exit point of the job and return control to the scheduler, we introduce a new system call, `TEE_exit_job()`, which notifies the scheduler of the job's completion and suspends execution until the next period, similar to a sleep operation. In Listing 5.1, we briefly show how one can mark the separation between the jobs of a task.

```
void ta_main_1_job(void)
{
    // control loop with 1 job
    for (;;) {
        int measurement = sensor();
        int result = compute(measurement);
        actuate(result);

        TEE_exit_job(0);
    }
}

void ta_main_3_jobs(void)
{
    // control loop with 3 jobs
    for (;;) {
        // job 1
        int measurement = sensor();
        TEE_exit_job(0);

        // job 2
        int result = compute(measurement);
        TEE_exit_job(0);

        // job 3
        actuate(result);
        TEE_exit_job(0);
    }
}
```

Listing 5.1: Example of a task divided into different numbers of jobs.

Policy loader and forwarder. To give the possibility to end-users to provision their policies to the Availability Monitor, we implement a policy loader and a policy forwarder. The policy loader is a user-level application executing in the normal world, which takes as argument a policy file. Its role is

to parse the policy and send it to the policy forwarder. As discussed in Section 2.2, communication between the secure and non-secure worlds is performed via a privileged instruction. Consequently, the policy loader cannot initiate this communication and directly forward the policy to the secure world. This is where the policy forwarder provides the necessary support. We implement the policy forwarder as a kernel module. It creates a character device that the policy loader will use to write the parsed policy. Once written, the policy forwarder is notified, after which it retrieves the provisioned policy and calls the Security Monitor in EL3. The Security Monitor then passes the policy to the Availability Monitor for further processing.

Passing DICE secrets. In order to pass the CDIs generated at each DICE flow to the next layer, we define shared memory regions between the different layers. While it is generally straightforward to do (i.e., we agree on a free, secure memory region and update the page table entries with the necessary mappings), the memory management subsystem in OP-TEE requires that all statically allocated secure RAM be contiguous. Hence, modifications in the memory layout of the TOS were required to allocate a shared memory region between itself and the DICE environment in the Secure Monitor.

The Attestation Engine. Following the recommendations of the implemented DICE profile [38] as well as the results of our empirical analysis (see Chapter 6), our Attestation Engine uses SHA-256 [67] as the cryptographic hash function, and EdDSA [17] on the Ed25519 Edward's twisted curve as the signature scheme. The generation of a public/private key pair in EdDSA relies on the provision of a random 32-byte number (the secret key), which will be projected on the curve to compute the public key. We leverage this mechanism to generate key pairs based on CDI [1] [38] instead, which in our case are 32-byte numbers, rather than random values polled from a random number generator. OP-TEE uses the LibTomCrypt library [46] as its cryptographic toolkit, which originally only supports the generation of EdDSA key pairs from a random number generator. We extend its functionality to implement our requirements, carefully following a well-established existing implementation [37]. Note that the secrecy of the private key then depends on the secrecy of the CDI.

The remote attestation. To enable communication with a remote verifier, we implement an attestation server in the normal world. Additionally, to bridge the Attestation Engine and the attestation server, we implement a pseudo-TA, that is, a privileged TA, which is scheduled by the attestation server whenever a remote verifier initiates the attestation process.

[1] Recall that the secret key is not actually derived directly from the CDI. As discussed in Section 4.2.3, it is first derived using a key derivation function, and the resulting value is then used as the seed.

Chapter 6

Evaluation

In this section, we present the results of the empirical analysis that we conducted on our implementation. We proceed by organizing our evaluation into three parts. First, we provide an evaluation of the performance metrics of the different components of the system. Then, we evaluate its security by (1) detailing where and by how much we increase the size of the TCB, (2) enhancing on the security issues that our design mitigates, and (3) by listing real-world attacks that our implementation is aware of. Finally, we provide two case studies, one that demonstrates the integration of our overall design into a toy setup based on a real-world industrial context, and the other, which implements an end-to-end remote attestation process.

6.1 Performance Evaluation

We divide the performance evaluation of our system in the same way we did our design. As a starting point, we will look further into the individual sections of secure schedulers. We then benchmark the scheduling latency incurred by our design on the critical tasks, and elaborate on the tools we use in the related section. Next, we assess the performance of the Attestation Engine, and compare the overhead induced by two well-known hashing algorithms in the context of our attestation procedure. Lastly, we evaluate the efficiency of our integration of secure real-time applications into the thread management subsystem of the trusted OS, and try to push those tasks to their limits.

We mainly run these evaluations by instrumenting the source code of our architecture. For consistency, we use the same physical counter, that is, `cntpct_el0`, to run all performance metrics. The physical counter is normally set up to trap when accessed from an unprivileged context. It suffices to enable the `PL0PCTEN` bit in the Counter-timer Kernel Control register (`cntkctl`) to let userland applications access it. We choose a timer-based approach instead of one based on CPU cycles, as we argue that it gives better insight in the context of real-time systems, as those systems

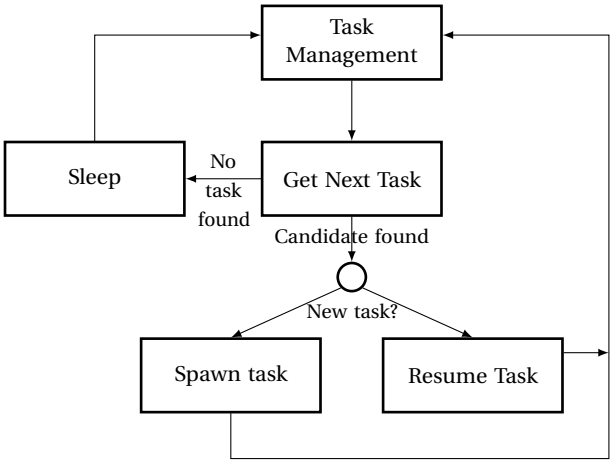


Figure 6.1: Functions of the secure scheduler and their interaction.

rely on timing requirements.

6.1.1 The Trusted Schedulers in Details

The Secure Scheduler

We profile the different sub-parts of the secure world scheduler. To help the reader visualize how the secure scheduler operates, we show a high-level perspective of its internals in Figure 6.1. This schematic illustrates the tasks and decisions of the scheduler. When ticking, the scheduler starts by performing necessary management on the task it potentially preempts, as well as its pool of tasks. Then it selects the next task to schedule in its ready queue. Depending on whether a candidate was found, it can either go idle and wake up at a later time, or proceed with the scheduling of the task. In the latter case, the task is either spawned or resumed, in which case the scheduler executes it until the next scheduling tick. Table 6.1 displays a broken-down performance analysis of that architecture. Apart from the first preemption of the task, the individual results show a reasonable overhead. As the task was the first to be scheduled on the system, the longer task management operation (13.6μs) could be caused by repetitive cache misses in the new memory regions (scheduler context, thread management context, etc.). The subsequent measurements for task preemption support this claim, as the overhead seems to stabilize to 3.9μs on average. Other than task preemption, the rest of the operations bear a fairly low impact on the overall performance of the system.

Operation	Description	Time (μs)
New task created	Task is loaded and ready for schedule	**2.8**
Task mgt	No task to preempt, general task management	0.3
Get next task	Newly created task found	1.2
Schedule task	Schedule the task for the first time	1.3
Will idle	Idle time incoming	**13.9**
Task mgt	Preempt the task that ran for the first time	13.6
Get next task	No task in running queue	0.3
Back from idle		**0.9**
Task mgt	No task to preempt, general task management	0.3
Get next task	The task is runnable again	0.2
Schedule task	The task will be resumed	0.4
Task preempted		
Task mgt	Preempt the task that ran for the nth time ($n > 1$)	3.9
...	...	

Table 6.1: Breakdown of the operations performed by the scheduler with their associated overhead. A single task is considered. We collected 200 samples.

Scheduling Latency

To perform our assessment on the scheduling latency, we in part use a common toolset for the evaluation of real-time performance. Specifically, we use the `cyclictest` tool of the `rt-tests` suite [42] to evaluate scheduling latency in Linux, alongside `stress-ng` [28], to stress the system while running the latency evaluation. `cyclictest` is a tool that periodically wakes up threads, which records the delay between the expected and the actual wake-up time. For the trusted side of the platform, we instrument the code of a sample real-time task to emulate our own `cyclictest`. When the task yields (i.e., sleeps), it expects to be rescheduled at the beginning of its next period; so when it is awakened by the secure scheduler, the task measures the gap with the beginning of its cycle.

We evaluate the latency of critical real-time tasks in different scenarios. We measure 1,000,000 samples for each evaluation. Figure 6.2 shows the baseline scenario in which the task runs on both cores with exclusive ownership, achieving a measured latency of 6μs in 99.4% and 99.7% of cases respectively. However, quintile extremes are as important in hard real-time systems, as they are the ones that may cause a task to miss its deadline, and bring potentially severe consequences. As Industrial Control Systems deal with control loops that can be scheduled at a 1kHz frequency, that is, with a period of 1ms, we consider acceptable measurements that do not go beyond 40-60μs. As we ought to evaluate our system under a lot of pressure from a potential adversary (either in the normal world, or in another real-time task), we run our secure world-centric `cyclictest` experiments in two settings.

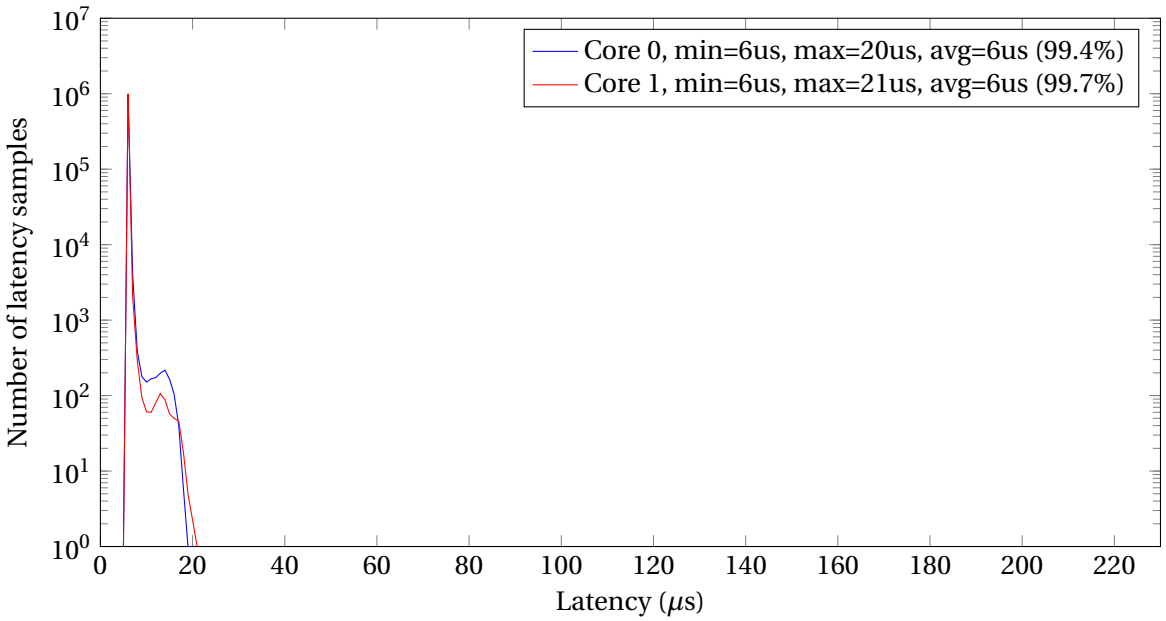


Figure 6.2: Cyclictest benchmark for 1M samples with exclusive ownership of cores.

First, we use the [28] tool to generate heavy traffic on the bus, which may interfere with secure transactions, such as the secure timer interrupt that our scheduler depends on. More specifically, we run it on both the application cores of our evaluation platform with 2 CPU stressors, 2 IO stressors, and 2 memory stressors. In the default implementation, OP-TEE enables foreign interrupts that generate a Fast Interrupt Request (FIQ) by the GIC on our board, and let the normal world preempt the Trusted OS to assert its interrupts. In our architecture, such a behavior is not sustainable as it would compromise the guarantees of availability. To motivate the deactivation of foreign interrupts while critical tasks are running, we enable their servicing during system calls only, and show the effects on the scheduling latency. We report the measurements from this experiment in Figure 6.3, during which the core was shared between the secure and non-secure contexts. The results show the impact of the stressors compared to the baseline in the shared core mode. Clearly, control is returned in the trusted world at a slower pace when the interconnected bus is flooded by the non-secure world. This is due to the secure timer interrupt being delayed by contention on the bus. Although the minimum and average latencies are similar, whether foreign interrupts are enabled or not, there is a clear performance impact in the extreme cases, as high as 260μs, representing more than 25% of the available execution time for tasks with a 1ms period. This is obviously not acceptable in safety-critical contexts, where the constraints for code size and speed are already restrictive enough.

In a second time, we investigate the impact of critical tasks on other critical tasks. In that spirit, we run two scenarios. The first one simulates a lower priority adversarial task with a demanding policy and code, in which the task never willingly yields, and declares an execution time equal to its period.

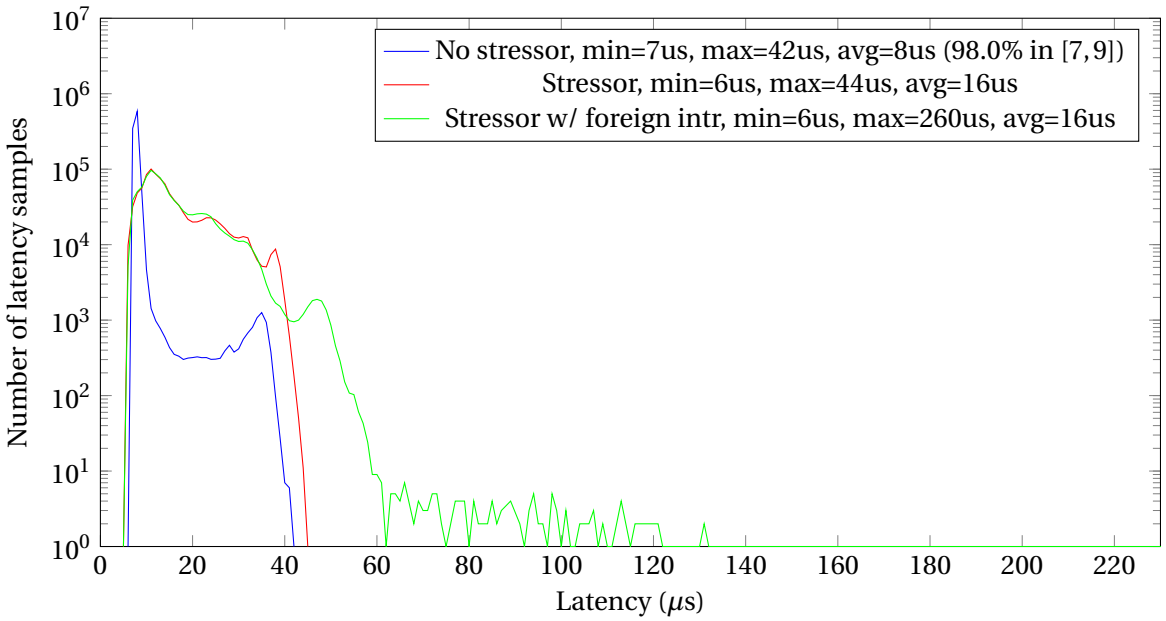


Figure 6.3: Cyclictest benchmark for 1M samples with shared ownership of cores. We measure here the impact the normal world can have on the secure scheduler.

In other words, the task has a utilization of 100% (recall Section 2.1.1) on the core. In the second variant of the experiment, we consider a higher priority task with a fixed execution time. We then measure the scheduling latency relative to that offset, to measure the overhead of task management and context switches. The measured task is defined as $T_1 = (2000, 1000, C_0, \emptyset)$. In the lower priority context, the adversarial task is defined as $T_2 = (2000, 1000, C_0, \emptyset)$ with $P(T_2) > P(T_1)$. In the higher priority context, the adversarial task is $T_3 = (1000, 1000, C_0, \emptyset)$ with $P(T_1) > P(T_3)$. We output the result of that experiment in Figure 6.4. We observe a slight overhead (a shift of 2–3 μs) when preempting a task based on its maximum execution time, compared to the baseline in Figure 6.2. This behavior is consistent with prior observations and can be explained by how absolute period- and relative execution time-based preemption are configured and handled internally. In the higher priority case, T_3 will be preempted based on the release of the next higher priority task, T_1 here. The absolute approach here tends to show better average results in minimizing scheduling latency. However, the high contention for resources shows a longer tail with delays going up to 57μs, double the observed values in the counterpart scenario.

Finally, as the secure tasks are not the only workload running on the system, we also evaluate the impact of our background scheduling approach for the non-critical applications. We run `cyclictest` in the normal world with tasks scheduled to run at a 10kHz frequency (100μs period), first without secure tasks running in parallel, and second alongside a task $T = (500, 100, C_0, \emptyset)$ a 100μs execution time. We display the results obtained in Figure 6.5. The results clearly show the 100μs time interval during which critical applications are scheduled, motivating the implementation

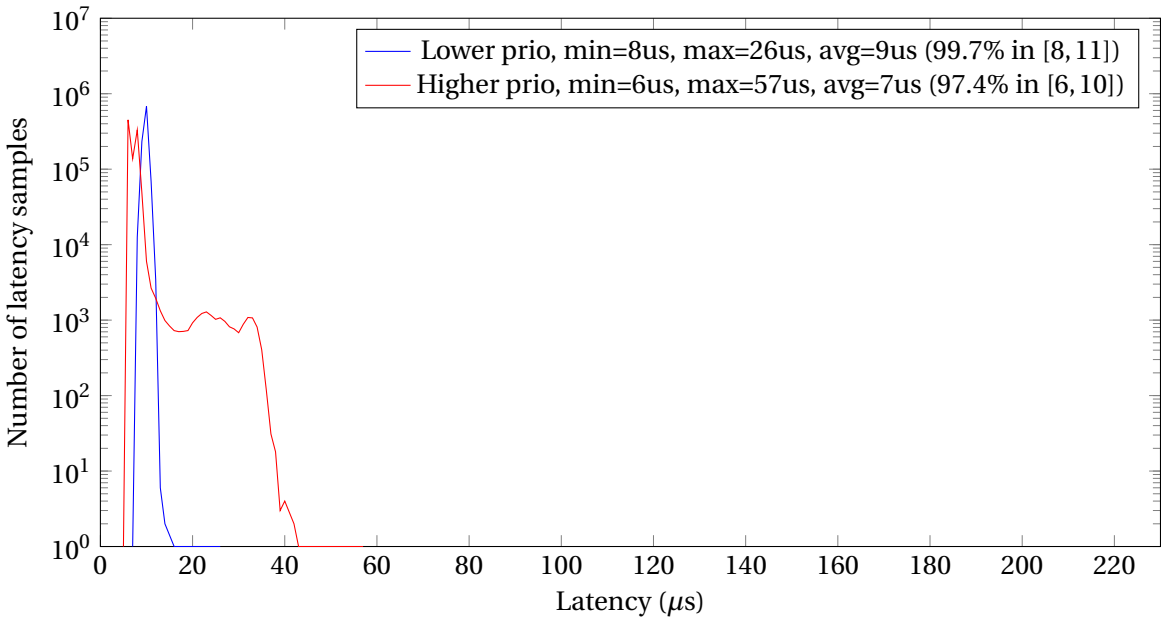


Figure 6.4: Cyclictest benchmark for 1M samples with high contention for resources. The measured task is defined as $T_1 = (2000, 1000, C_0, \emptyset)$. In the lower priority context, the adversarial task is defined as $T_2 = (2000, 1000, C_0, \emptyset)$ with $P(T_2) > P(T_1)$. In the higher priority context, the adversarial task is $T_3 = (1000, 1000, C_0, \emptyset)$ with $P(T_1) > P(T_3)$.

of real-time tasks in the secure environment, to benefit from the offered availability guarantees.

6.1.2 The Attestation Engine

In this section, we evaluate the performance of our Attestation Engine and compare two different implementations, one using a SHA-256 hashing algorithm, and the other using SHA-512. We compare the impact of these two hashing schemes on the overall performance of the AE.

Hashing and Signing. To reliably compare the two implementations, we run our evaluation setup three times. The first run does not generate any claim and therefore consists of the signing part only, where a buffer of null bytes is signed. This allows us to isolate the signing part from the hashing part of the attestation mechanism. In the two subsequent runs, the attestation is performed against a medium-sized TA (73,792 bytes) and a large TA (1,122,368 bytes). This allows us to get an idea of the scalability of the methods. We report the results of our evaluation in Figure 6.6 and discuss the results below. Our evaluation indicates a relatively poor scalability of the SHA-512 algorithm. Note, however, that this is very specific to the underlying implementation and computing capabilities of the platform. Although a hardware-based implementation could potentially procure better performance and security, a lot of devices still in use today do not benefit from hardware

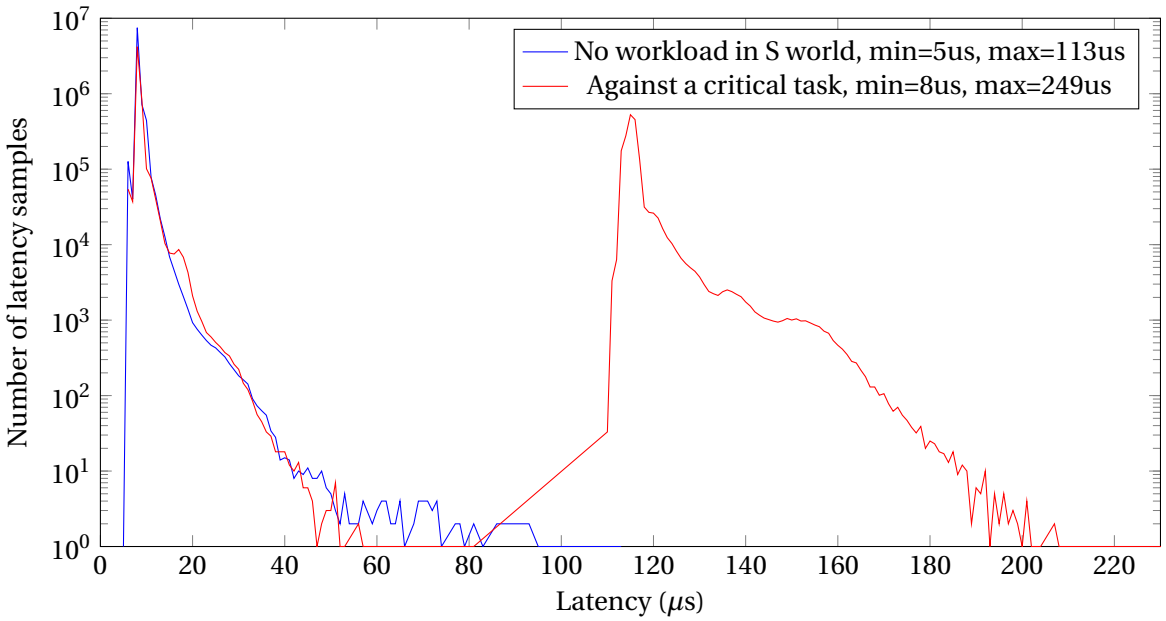


Figure 6.5: Impact of secure real-time applications on workload executing in the normal world. With and without a parallel critical task $T = (500, 100, C_0, \emptyset)$. Both experiments ran for 900 seconds, accounting for 8,999,999 samples in the baseline case, and 7,199,984 in the second case.

cryptographic engines. A software implementation presents the advantage of portability. However, real-time applications typically have a much smaller footprint than the large TA evaluated here, and the relatively more adequate memory footprints used in our setup suggest that the performance of the attestation is primarily limited by the signature mechanism.

6.1.3 The Critical Real-Time Tasks

We now assess the performance related to real-time tasks. First, we provide an analysis of the different life cycles of tasks. Then, we run metrics on the maximum frequency at which a critical application can be scheduled, in order to determine an upper bound on the frequency at which our system can schedule those applications.

Critical task life-cycle breakdown. We present the results of our measurements in Table 6.2. When a task is first allocated, it needs to be loaded into memory. This is a costly process which incurs a high overhead (15ms in our evaluation) compared to the other operations of the life cycle. However, we argue that this performance impact is acceptable, as it is only performed once for the entire lifetime of the application. Then, the first time a task is scheduled, it needs to go through the process of thread and resource allocation (i.e., opening of an OP-TEE session). This process, measured to be on average 50μs, could drastically impact the scheduling latency if it were to be repeated for

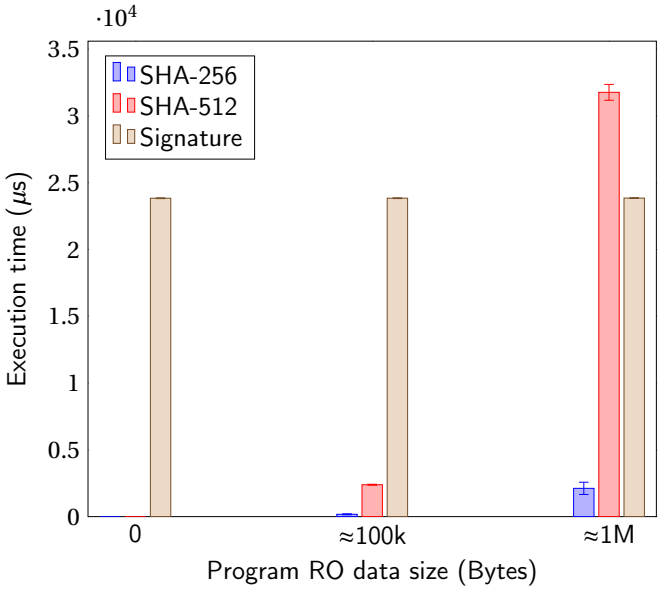


Figure 6.6: Performance of the attestation mechanism for three different tasks memory footprints. The attestation mechanism is as follows: program code and RO data are hashed, task fingerprint is constructed, both are signed together. The signature scheme is run twice for each size, once with SHA-256 as the hash algorithm to measure the task's memory, and once with SHA-512. The exact size of the memory footprints are 0 bytes (baseline), 73792 bytes, and 1122368 bytes.

Operation	**Description**	**Time (μs)**
Allocation	Retrieve the task binary from memory and load it	15115 ± 14
First entry to exit	First entry is done via a specialized `TEE_InvokeCommand()`	50 ± 1
Re-entries to exit	Subsequent re-entries are done by restoring context	4 ± 0

Table 6.2: Breakdown of the lifecycle operations of the critical real-time tasks with associated overheads. 10 samples were collected.

each schedule. This is the reason we decided on another mechanism to reschedule the tasks, which consists of never returning from the invocation and only performing context switches from there on. All subsequent re-entries into the task context incur a much smaller impact on the scheduling latency. Reported measurements indicate that a round-trip of context switches (i.e., scheduler to application and back) is performed in 4μs.

Frequency of the tasks. In order to empirically determine the maximum frequency at which real-time applications can be scheduled on our system, we instrument a critical task that originally does nothing and yields instantaneously. The measurements collected, reported in Figure 6.7, indicate that a real-time applications can be scheduled with a frequency of about 110,000Hz, that is, at a period of 9μs. This shows the practicality of our solution in industrial contexts, where the period of protection functions, while still rapid, is much slower.

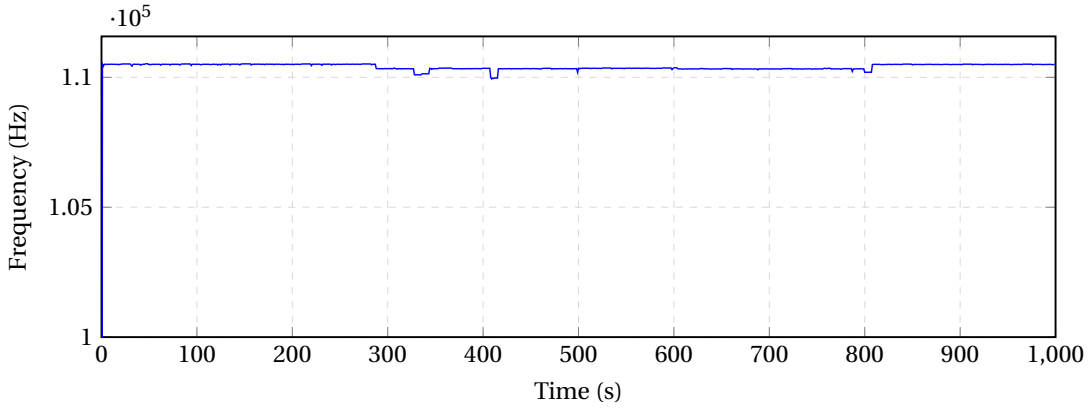


Figure 6.7: Maximum frequency of a simple real-time application over a 1000 seconds period. The task yields after each tick and is the only trusted RT apps scheduled on the system.

6.2 Security Evaluation

We now evaluate the security aspect of our system. First, we measure the impact of our additions on the trusted computing base (TCB). Then, we continue with an analysis of different attack cases, and discuss how our security architecture mitigates them.

6.2.1 TCB

In order to determine the increase in the size of the TCB imposed by our system, we count and report the number of lines of code (LoC) that we added. We use `cloc` [3] to count the number of LoCs and give a detailed summary of the added LoCs and their associated component in Table 6.3. Overall, we increase the size of the TCB with about 773 LoCs in U-Boot SPL, which is discarded when entering the TF-A Secure Monitor in EL3, 614 in TF-A, and 2069 in OP-TEE. This accounts for less than a 1% increment in total.

6.2.2 Security Analysis

In this section, we assess our system against adversarial scenarios using the MITRE ATT&CK Matrix for ICS [53], showing how our design mitigates concrete attack scenarios. Here, we try not to assess the mitigation of threats already discussed in Chapter 4, and focus on more general situations.

Hardcoded credentials. An adversary may exploit hardcoded credentials stored in software to gain persistent access to a system's assets, which might be hard or impossible to revoke, should the secrets be leaked. With the integration of DICE into our design, only a single secret is persistently

Description	LoC
U-Boot SPL	**773**
DICE Engine	572
DICE Integration	124
Firewall Driver	77
Trusted Firmware-A	**614**
World Scheduler	470
DICE Service	123
OP-TEE SPD	21
OP-TEE	**2069**
Scheduler	953
Device Agents	322
Secure Drivers	117
Attestation Engine	325
Memory Management	5
Thread Management	155
TA Management	24
OP-TEE Entry	130
Crypto Library	38

Table 6.3: Added lines of code per project to measure the TCB increase in LoCs.

stored on the device. However, this secret is only accessible for a short period of time during the boot stages, after which it is made inaccessible until the next reset. Furthermore, this secret is only used as a starting point to generate hardware-software bound credentials, and is never used in any other form. This makes revocation of keys simple, and leakage of long term secrets unlikely.

Program modifications. An adversary might exploit bugs or misconfigurations in the system to add or modify critical programs on the system. We leverage the TrustZone hardware abstraction layer to securely isolate the resources of safety-critical applications. We also provide mechanisms to verify that the correct version of a program is loaded, as well as its provisioned real-time policy, thereby detecting and protecting real-time tasks from illegal modifications.

Denial of Service. A remote or local adversary may attempt to disrupt the availability of the system by overwhelming its resources. During our evaluation, we simulated such a scenario by stressing the CPU, memory system, as well as communication with IO peripherals. The results reported in Figure 6.3 show the efficacy of our system in preventing such attacks.

Loss of protection. Given sufficient access, a malicious actor could manipulate the system to disable protection functions, resulting in failure to react to faulty processes in an industrial plant.

The consequences could be many, and as such, protecting against this scenario is one of the core motivating aspects of our work. By running critical tasks in the secure environment under the protection of the Availability Monitor, critical functions are ensured continuous up-time for as long as the system is live.

Rootkit. Rootkits are a type of malware that is generally deployed at the root of systems, such as the untrusted operating system. The level of privilege at which they run makes them hard to detect and able to hide the presence of other malicious entities on the system. We consider and protect against a fully compromised normal world OS, even from system startup. Indeed, critical tasks and their policies can be pre-loaded and run even before the normal world has started. Given those protections and those enforced by TrustZone at the hardware level, a malicious untrusted kernel can therefore not access or prevent computation in the secure world. However, we highlight that our architecture does not protect normal world operations, nor does it prevent the spread of malicious software within it.

IO image. Attackers could capture the state of an IO image, which holds all state and potentially past measurements from a device. They might alter configuration values, which could have direct consequences on the relying protection functions, or use it to plan future attack stages. Device Agents mitigate this attack vector by enforcing strict spatial isolation on the secure peripheral they manage.

6.3 Case Studies

In order to further evaluate the feasibility of real-world scenarios on our system, we now present two case studies for which we implemented a solution.

6.3.1 A Sample Industrial Setup

We implement a sample industrial control system in which both the protection function and a data acquisition entity, such as a SCADA, can retrieve data from the device. The integration of a reader external to the secure world shall not in any case disturb execution in the protected environment. To mimic this sample environment, we first implement an IO image that periodically, i.e., in real-time, reads data from a device and stores it in a ring buffer. Then, we implement a real-time task whose purpose will solely reside in reading the last entry in the IO image's buffer, performing an arbitrary control computation on this data, and outputting a result to an actuator. In this case study, the sensor is an external timer with its Device Agent and secure driver, and the actuator is a simple memory buffer. We implement an external reader in the normal world, which can query an arbitrary

number of entries from the IO image. A real-world application of this external reader could be that of a SCADA system retrieving past measurements to display in a control room. In that case, the SCADA polls the data in an aperiodic way, or at a low frequency compared to the real-time protection function. An overview of this architecture is presented in Figure 6.8. We schedule the real-time subsystem on one core with both tasks running with a 1000μs period and 500μs execution time, and the SCADA interface on the other core. Our evaluation shows no missed deadlines of the critical periodic tasks (i.e., the IO image and the protection function), while allowing a third party to also read from the device.

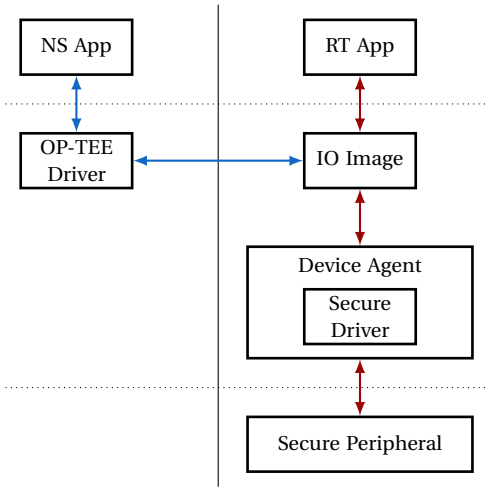


Figure 6.8: Overview of the architecture used for the sample industrial setup of the case study presented in Section 6.3.1. Red (resp. blue) arrows represent secure (resp. non-secure) communication.

6.3.2 End-to-End Remote Attestation

Our second case study presents an end-to-end implementation of the remote attestation procedure, which we illustrate in Figure 6.9. To that end, we implement an attestation server in the normal world, which allows a remote verifier to contact our Attestation Engine. The attestation server is built as a Client Application (CA), that is, a normal world application communicating with a Trusted Application (TA). To bridge the CA and the AE, we implement an attestation bridge, that simply consists of a Pseudo TA (i.e., a privileged TA). Finally, we implement an attestation client on a separate device to initiate the procedure. We connect our system to the network and run the attestation server on a core, while a critical real-time task runs on the other core. The attesting client, which can be perceived as a SCADA system, initiates the attestation procedure by sending a challenge and the unique identifier of a target real-time application. The attestation server receives the payload, and forwards it to the attestation bridge. The attestation bridge then queries the AE, which retrieves the running task data and signs it to form the evidence. The evidence is then sent back to the remote verifier through the same route. The verifier, which knows the public key of

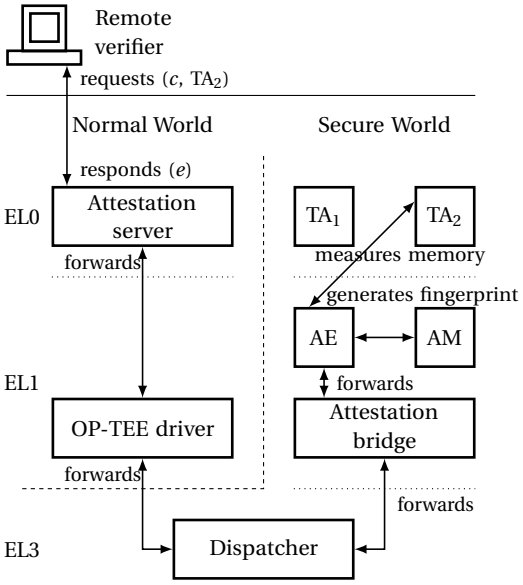


Figure 6.9: Overview of the end-to-end attestation process between a remote verifier and our system detailed in Section 6.3.2.

the Attestation Engine, and has access to golden measurements of the system, that is, valid hashes, authenticates the received evidence and proceeds to verify the claims against the reference values. We evaluate this setup in a normal context (i.e., everything runs as expected), in another context where the TA experienced delays, and in an adversarial context where a third party would try to tamper with the evidence. The results of this evaluation show the practicality of our design for remote attestation, and show that an invalid claim or tampered evidence can be detected by a remote verifier.

Chapter 7

Discussion

We examine here the portability of our work to other Instruction Set Architectures (ISA), discuss observed limitations in the current design, and provide potential directions for future work.

7.1 Portability of the Design to Other ISAs

Although Arm remains the platform of choice for the implementation of our objectives (recall Section 3.1, other architectures have emerged in recent years, offering a hardware abstraction layer that delivers security guarantees similar to TrustZone. One such architecture is RISC-V. RISC-V provides the Physical Memory Protection (PMP) registers, which allow for the dynamic partitioning of resources among the different privilege levels. To isolate DMA-driven peripherals, a hardware component called IOPMP is used. Keystone [44] leverages the PMP registers to provide a framework for the creation of dynamic trusted execution environments. It readily supports provisioning of device keys for remote attestation, performed by its Security Monitor (SM), which also handles physical resources and supports the delegation of interrupts. Such mechanisms indicate support for enforcing peripheral isolation. The SM can additionally provide secure timers and sealed storage. We also note that OP-TEE implements support for the RISC-V architecture. Overall, these capabilities confirm the availability of the requirements to port our design to RISC-V systems.

7.2 Limitations

In this section, we acknowledge and discuss the limitations and issues we observed in our current design.

Developer effort. Our design allows system designers to provide the requirements of their tasks, as well as to decide which devices should be protected or not, depending on their requirements. Consequently, the required drivers and Device Agents should be designed in accordance with those requirements. While this allows for a high degree of flexibility in how the Availability Monitor behaves, it induces a higher design time engineering effort. However, since our attestation engine verifies the behavior of the AM, we argue that a faulty configuration can be easily detected.

Information leakage. Since the policies are sent in clear via the normal world, a malicious entity with knowledge of the mapping between unique identifiers and the critical components protected by the real-time task could infer information about the state of the system, in particular, the presence or absence of specific control loops. This could give information on potential holes in the security of the industrial plant. Furthermore, our remote attestation mechanism does not authenticate the verifier. To initiate the attestation procedure, a remote verifier sends a challenge with the identifier of a target task to the attestation engine. The returned evidence is constant for all tasks that could not be found in the AM memory, that is, tasks that are not loaded on the system. To give a simple example, the response could be the hash of a constant value, such as a string of null bytes (since the task was not found), accompanied by the signature of that hash. A malicious verifier, or any malicious entity contacting the attestation server, could use the remote attestation service as a feature-detection oracle. Resulting in the same aforementioned issues of theft of operational information.

7.3 Future work

Leveraging heterogeneous platforms. Arm SoCs, among others, often provide heterogeneous configurations where different types of processors are interconnected on the same platform. For instance, the SoC on which we implemented and evaluated our design features Cortex-A cores (high-end embedded application cores), commonly used to run a Linux/OP-TEE system, a Cortex-M core (low-end embedded core) for the security controller, and Cortex-R cores (real-time cores) to run safety-critical tasks. Our current design does not leverage this heterogeneity and focuses solely on application cores. Scheduling complex real-time tasks on heterogeneous systems is an emerging topic, that brings interesting challenges in terms of scheduling and security, as well as benefits, such as enhanced energy efficiency by optimally scheduling tasks on different processors based on their computational requirements.

Arm Confidential Compute Architecture. Although not yet available on off-the-shelf hardware, we motivate the investigation of Real-Time TEEs on environments enabled by the Arm Confidential Compute Architecture (CCA) [14]. The Granule Protection Tables (GPT), that is, the isolation primitives in Arm CCA, are part of the specification of Arm v9 processors, which greatly eases portability across multiple hardware platforms. Furthermore, when an illegal access is detected

during a Granule Protection Check (GPC), a synchronous fault, the so-called Granule Protection Fault (GPF), is generated. The synchronous nature of those faults allows for a facilitated reconstruction of faulty instructions, such as read or write operations to a device, paving the way for transparent isolation of secure devices from the point of view of the non-secure world. As a result, requests to access a secure device from the NS world could be transparently performed upon a faulty access, greatly increasing scalability and reducing the amount of design-time engineering in the normal world. Although not peer-reviewed at the time of writing, we refer the reader to [18] for additional details on such a mechanism.

Chapter 8

Related Work

Although contributions to specifically provide availability guarantees through trusted scheduling [4, 51, 85, 86, 89], or remotely attest execution [23, 29, 52, 54, 56, 71, 87] have gained considerable attention in recent years, coupling TEE with real-time systems [4, 34, 43, 51, 58, 59, 85, 86, 89] has been an active research topic for several years. In this section, we first discuss Trusted Execution Environments that, although not considering availability, are still relevant and related to our work, as they address necessary requirements to enable availability. We then consider TEEs that address the challenges of availability in real-time systems and categorize them into two groups, depending on whether they rely on customized hardware. Finally, we will highlight research efforts tackling attestation of availability. For all these designs, we discuss the problem they solve and how our contribution differs or builds upon them.

TEE for cyber-physical systems. The ANDIX research OS [34] is a TEE for ICS, providing integrity and confidentiality to secure critical tasks. Its source code was made publicly available to foster research on trusted environments for industrial control systems. Our work is complementary in the sense that it extends the provided security guarantees with availability. M2MON [43] is a security monitor developed as a microkernel, that monitors MMIO accesses in unmanned vehicles to enforce access control over critical hardware resources. It achieves this isolation by mapping critical devices to the trusted environment and modifying each direct access in the normal world by calls to the monitor. A similar mechanism is employed by the non-secure part of our split drivers and that of [64, 84–86], though at a coarser granularity.

Availability guarantees with hardware modifications. In their work, Masti et al. [51] introduced the concept of trusted scheduling, where TEEs are leveraged to provide guaranteed scheduling. They laid down the hardware requirements for so-called trusted scheduling and paved the way for future research in that area. However, their architecture requires that the set of applications

running on the system be pre-defined and not meant to change at runtime. Aion [4] is a security architecture that extends the Sancus TEE [55], and builds on the foundation of [51]. They provide similar security guarantees on a single core, while proposing an open system, and motivating the need for a remote verifier to be able to attest to the guaranteed progress of real-time tasks. They offer properties that closely match the challenges we seek to overcome, yet they do not provide a design compatible with off-the-shelf hardware, an important requirement of our work. Hora [89] is a subsequent work providing similar security guarantees while also preventing untrusted applications from compromising the drivers of shared peripherals. It leverages the heterogeneity of Arm platforms to schedule the life-critical applications of a smartphone on real-time cores, while the rest of the system runs on the application cores. Similar to our approach, Hora is intended to exploit the capabilities of multicore platforms. To enable resource sharing, they employ a mechanism similar to the Dynamic Priority Ceiling Protocol (D-PCP) [63], in which requests to access a peripheral are sent to an I/O domain, which ensures exclusive access and performs the actual device operations on their behalf. This mechanism is mainly used on distributed memory multicore platforms, that is, when each core has its own private memory. Since our design focuses on shared-memory multiprocessors, we implement a version of the multicore PCP adapted to this architecture. However, the implemented design lacks the flexibility to configure the period of real-time tasks, limiting cases where the system runs tasks that need to execute at different intervals of time.

Availability guarantees on COTS hardware. Similarly to this work, Mr-TEE [85], NetReach [84], RT-TEE [86], and the work of Röckl et al. [64] all build on top of the TrustZone hardware abstraction layer. Furthermore, they all leverage a split-driver design for the drivers of shared peripherals. While [84] and [64] specifically focus on the availability of the network stack, which was not addressed before, [85] and [86] address a broader scope, without providing deep details on the implementation of particular drivers. Although the four of them focus on resource availability, with an emphasis on trusted scheduling for the two general approaches, they leave the attestation of execution outside the scope of their work. To enable trusted scheduling, RT-TEE implements a policy-based, event-driven, hierarchical scheduler, which we use as the foundation for the design of our secure scheduler. Both real-time TEE provide similar security guarantees but follow a different strategy in the trade-off between TCB minimization and developer effort. RT-TEE's main focus is on minimizing the size of the TCB, which is accomplished by careful design time analysis, control flow integrity instrumentation, software fault isolation techniques, and taint analysis for driver debloating. While it allows them to achieve their goals, it requires significant design time engineering in both the secure and non-secure worlds, and ultimately makes it a closed, single-tenant system. Furthermore, it does not allow interrupts to be assigned to the secure world, limiting use cases for real-time tasks. On the other hand, Mr-TEE puts the focus on minimizing the developer effort, and introduces a novel mechanism to share interrupts between the trusted and untrusted domains. However, its secure scheduler only runs on a single core, which, while reducing complexity in the TCB, could rapidly become a limitation in some industrial contexts where the frequency of control loops can be as fast as 1kHz. Our work differs from theirs in that we aim to provide remotely attestable availability,

and let our real-time tasks take full advantage of the multiprocessor platform. Moreover, we intend to offer the end-user at least the same degree of flexibility as Mr-TEE.

Attestation of availability. [2] is a TrustZone-based system that addresses the problem of software-based attestation integrity at runtime by attesting the Control-Flow Integrity (CFI) of a targeted program. It acknowledges its lack of support for Data Flow Integrity (DFI), which it reports as future work. [71] introduces the concept of Operation Execution Integrity (OEI) and proposes a software-based approach to attest this property, leveraging TrustZone. They do so by attesting both CFI and DFI. In parallel, [56] introduces a similar security property, which is named Proof of eXecution (PoX). [56] presents the APEX architecture, which relies on hardware modifications to provide remotely attestable PoX for a target application. In order to provide PoX, their design requires atomic (i.e., non-interrupted) execution of the target application, which is not applicable in a real-time system where a job can be preempted to schedule a higher priority one. [23] is a first attempt to remediate this issue by extending the APEX architecture to enable a target real-time task to observe interrupts. However, it still blocks those generated from outside the context of the task, thereby leaving the issue of preemption open. To bridge the gap between PoX and preemption in real-time systems, [54] proposes Real-Time PoX (RT-PoX), and elicits the necessary conditions to achieve this property. [54] is foundational to our design, as it establishes a framework of the necessary conditions to achieve RT-PoX. [87] introduces the concept of Real-Time Mission Execution Integrity (RMEI), a property similar to RT-PoX, and provides a system to remotely verify that this property holds. RMEI is said to hold if both spatial (CFI, DFI) and temporal (real-time timing constraints) behaviors are respected. The system is continuously measured throughout the entire mission, measurements are encrypted, and then stored until the end of the mission, where a verification engine verifies the stored values. In order to reduce the performance overhead induced by such a mechanism, the attested system is split in critical and non-critical sections. This requires a non-negligible amount of engineering work during the design phases to define policies, compartmentalize, and instrument the program. While this mechanism is well justified in the context of a closed system where it is performed only once over the system's lifetime, it may face challenges in scaling to open systems similar to our work. Other research efforts [29, 52] address attestation of execution, but leverage a hardware TPM [79], therefore being outside the scope of our study. Although all these works tackle a similar problem, which is runtime attestation of unaltered execution, they solely focus on attestation and do not provide guaranteed availability.

Chapter 9

Conclusion

In this work, we introduced a novel architecture that bridges attestation of execution with guarantees of real-time availability, even in the presence of a fully compromised normal world OS present from system startup. By leveraging commodity, resource-efficient Arm multiprocessor platforms, we enable both predictable and verifiable trust for critical industrial infrastructures, where the end-user can run workload from multiple mutually distrusting but equally privileged vendors. Through separation of powers inside our architecture, we reduce the risk of centralized trust and improve the resilience of the system against misconfiguration and adversarial interference. To ensure real-time availability, our design proposes an Availability Monitor that enforces user-defined policies, under the supervision of our attestation engine. Our prototype implementation demonstrates that our architecture can support high-frequency real-time task execution (up to 110 kHz in a theoretical scenario) with relatively low scheduling latency (around 6 μs in the best case), while also preserving confidentiality and integrity of proprietary code and data. The attestation engine reliably produces evidences of correct scheduling, execution, and configuration, without compromising system availability. To the best of our knowledge, this is the first architecture to bridge trusted real-time scheduling with attestation of execution, especially while leveraging multicore commodity platforms.

Bibliography

[1] ABB. *IRB 8700*. URL: https://new.abb.com/products/robotics/robots/articulated-robots/irb-8700.

[2] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. "C-FLAT: Control-Flow Attestation for Embedded Systems Software". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 743–754. ISBN: 9781450341394. DOI: 10.1145/2976749.2978358. URL: https://doi.org/10.1145/2976749.2978358.

[3] AlDanial. *cloc*. Version 2.06. 2025. URL: https://github.com/AlDanial/cloc.

[4] Fritz Alder, Jo Van Bulck, Frank Piessens, and Jan Tobias Mühlberg. "Aion: Enabling Open Systems through Strong Availability Guarantees for Enclaves". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 1357–1372. ISBN: 9781450384544. DOI: 10.1145/3460120.3484782. URL: https://doi.org/10.1145/3460120.3484782.

[5] Android Open Source Project. *Trusty TEE*. 2025. URL: https://source.android.com/docs/security/features/trusty.

[6] Apple Inc. *Hardware Root of Trust*. 2025. URL: https://security.apple.com/documentation/private-cloud-compute/hardwarerootoftrust.

[7] Arm. *ARM CoreLink TZC-400 TrustZone Address Space Controller Technical Reference Manual*. Version r0p1. 2013. URL: https://developer.arm.com/documentation/ddi0504/c.

[8] Arm. *ARM Security Technology Building a Secure System using TrustZone Technology*. Version Revision C. 2008. URL: https://developer.arm.com/documentation/PRD29-GENC-009492/c.

[9] Arm. *ARM1176JZF-S Technical Reference Manual*. Version Revision H. 2004. URL: https://developer.arm.com/documentation/ddi0301/h/.

[10] Arm. *CoreLink TrustZone Address Space Controller TZC-380 Technical Reference Manual*. Version r0p1. 2008. URL: https://developer.arm.com/documentation/ddi0431/c/.

[11] Arm. *Learn the architecture - AArch64 Exception Model*. Version 1.3. 2022. URL: https://developer.arm.com/documentation/102412/0103.

[12] Arm. *Learn the architecture - Generic Interrupt Controller v3 and v4, Overview*. Version 3.2. 2021. URL: https://developer.arm.com/documentation/198123/0302.

[13] Arm. *Learn the architecture - Generic Timer*. Version 1.1. 2022. URL: https://developer.arm.com/documentation/102379/0101.

[14] Arm. *Learn the architecture - Realm Management Extension*. Version 1.1. 2021. URL: https://developer.arm.com/documentation/den0126/0101.

[15] Arm. *Learn the architecture - TrustZone for AArch64*. Version 1.2. 2020. URL: https://developer.arm.com/documentation/102418/0102.

[16] Arm. *SMC Calling Convention*. Tech. rep. DEN0028D. Version v1.3. Arm, 2021.

[17] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. "High-speed high-security signatures". In: *Journal of Cryptographic Engineering* 2.2 (2012), pp. 77–89. ISSN: 2190-8516. DOI: 10.1007/s13389-012-0027-1. URL: https://doi.org/10.1007/s13389-012-0027-1.

[18] Andrin Bertschi, Supraja Sridhara, Friederike Groschupp, Mark Kuhne, Benedict Schlüter, Clément Thorens, Nicolas Dutly, Srdjan Capkun, and Shweta Shinde. "Devlore: Extending Arm CCA to Integrated Devices — A Journey Beyond Memory to Interrupt Isolation". arXiv preprint arXiv:2408.05835. 2024. DOI: 10.48550/arXiv.2408.05835. arXiv: 2408.05835 [cs.CR]. URL: https://arxiv.org/abs/2408.05835.

[19] Henk Birkholz, Dave Thaler, Michael Richardson, Ned Smith, and Wei Pan. *Remote ATtestation procedureS (RATS) Architecture*. RFC 9334. Jan. 2023. DOI: 10.17487/RFC9334. URL: https://www.rfc-editor.org/info/rfc9334.

[20] Björn B. Brandenburg and James H. Anderson. "A Comparison of the M-PCP, D-PCP, and FMLP on LITMUSRT". In: *Principles of Distributed Systems*. Ed. by Theodore P. Baker, Alain Bui, and Sébastien Tixeuil. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 105–124. ISBN: 978-3-540-92221-6.

[21] Giorgio C. Buttazzo. "Rate monotonic vs. EDF: judgment day". In: *Real-Time Syst.* 29.1 (Jan. 2005), pp. 5–26. ISSN: 0922-6443. DOI: 10.1023/B:TIME.0000048932.30002.d9. URL: https://doi.org/10.1023/B:TIME.0000048932.30002.d9.

[22] Charly Castes, Adrien Ghosn, Neelu S. Kalani, Yuchen Qian, Marios Kogias, Mathias Payer, and Edouard Bugnion. "Creating Trust by Abolishing Hierarchies". In: *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. HOTOS '23. Providence, RI, USA: Association for Computing Machinery, 2023, pp. 231–238. ISBN: 9798400701955. DOI: 10.1145/3593856.3595900. URL: https://doi.org/10.1145/3593856.3595900.

[23] Adam Caulfield, Norrathep Rattanavipanon, and Ivan De Oliveira Nunes. “ASAP: reconciling asynchronous real-time operations and proofs of execution in simple embedded systems”. In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. DAC ’22. San Francisco, California: Association for Computing Machinery, 2022, pp. 721–726. ISBN: 9781450391429. DOI: 10.1145/3489517.3530550. URL: https://doi.org/10.1145/3489517.3530550.

[24] David Cerdeira, José Martins, Nuno Santos, and Sandro Pinto. “ReZone: Disarming TrustZone with TEE Privilege Reduction”. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 2261–2279. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/cerdeira.

[25] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. “SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1416–1432. DOI: 10.1109/SP40000.2020.00061.

[26] CH Info. *Separation of powers*. 2025. URL: https://www.ch-info.swiss/en/edition-2025/direkte-demokratie/gewaltenteilung.

[27] Chia-mei Chen and Satish Tripathi. “Multiprocessor Priority Ceiling Based Protocols”. In: *Technical Report CS-TR-3252, University of Maryland* (Apr. 1994).

[28] Colin Ian King and the stress-ng Development Community. *stress-ng (system stress testing tool with extensive stressors)*. GitHub repository. Tool designed to stress test various physical subsystems and kernel interfaces across many platforms. 2025. URL: https://github.com/ColinIanKing/stress-ng.

[29] Victor Costan, Ilia Lebedev, and Srinivas Devadas. “Sanctum: Minimal Hardware Extensions for Strong Software Isolation”. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 857–874. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan.

[30] *Das U-Boot Source Tree*. GitHub repository. Bootloader for embedded boards: PowerPC, ARM, MIPS, and more. 2025. URL: https://github.com/u-boot/u-boot.

[31] R.I. Davis and A. Burns. “Hierarchical fixed priority pre-emptive scheduling”. In: *26th IEEE International Real-Time Systems Symposium (RTSS’05)*. 2005, 10 pp.–398. DOI: 10.1109/RTSS.2005.25.

[32] Robert I. Davis and Alan Burns. “A survey of hard real-time scheduling for multiprocessor systems”. In: *ACM Comput. Surv.* 43.4 (Oct. 2011). ISSN: 0360-0300. DOI: 10.1145/1978802.1978814. URL: https://doi.org/10.1145/1978802.1978814.

[33] Morris J. Dworkin. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. Tech. rep. NIST Special Publication 800-38D. National Institute of Standards and Technology, 2007. DOI: 10.6028/NIST.SP.800-38D.

[34] Andreas Fitzek, Florian Achleitner, Johannes Winter, and Daniel Hein. "The ANDIX research OS — ARM TrustZone meets industrial control systems security". In: *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*. 2015, pp. 88–93. DOI: 10.1109/INDIN.2015.7281715.

[35] Linux Foundation. *Linux Kernel*. Version 6.14. 2025. URL: https://git.kernel.org/.

[36] GlobalPlatform. *GlobalPlatform TEE Internal Core API Specification, v1.3.1*. Specification GPD_SPE_010. GlobalPlatform, 2021. URL: https://globalplatform.org/specs-library/tee-internal-core-api-specification/.

[37] Google. *BoringSSL: A Fork of OpenSSL Designed to Meet Google's Needs*. 2015. URL: https://boringssl.googlesource.com/boringssl/+/master/.

[38] Google. *Open Profile for DICE*. Specification. Version v2.5. Google, 2025. URL: https://pigweed.googlesource.com/open-dice/+/HEAD/docs/specification.md.

[39] Andy Greenberg. *Hackers Remotely Kill a Jeep on the Highway–With Me in It*. 2015. URL: https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/.

[40] Haoyang Huang, Fengwei Zhang, Shoumeng Yan, Tao Wei, and Zhengyu He. "SoK: A Comparison Study of Arm TrustZone and CCA". In: *2024 International Symposium on Secure and Private Execution Environment Design (SEED)*. 2024, pp. 107–118. DOI: 10.1109/SEED61283.2024.00021.

[41] Intel Corporation. *Intel Security Essentials – Built-in Foundation with Security at the Core*. Solution Brief. Intel Corporation, 2025. URL: https://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/intel-security-essentials-solution-brief.pdf.

[42] jlelli and the rt-tests Development Community. *rt-tests – Real-Time Test Utilities*. GitHub repository. Includes source for cyclictest, a real-time latency measurement tool. 2025. URL: https://github.com/jlelli/rt-tests.

[43] Arslan Khan, Hyungsub Kim, Byoungyoung Lee, Dongyan Xu, Antonio Bianchi, and Dave (Jing) Tian. "M2MON: Building an MMIO-based Security Reference Monitor for Unmanned Vehicles". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 285–302. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/khan-arslan.

[44] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. “Keystone: an open framework for architecting trusted execution environments”. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys ’20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387532. URL: https://doi.org/10.1145/3342195.3387532.

[45] Mengyuan Li, Yuheng Yang, Guoxing Chen, Mengjia Yan, and Yinqian Zhang. “SoK: Understanding Design Choices and Pitfalls of Trusted Execution Environments”. In: *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. ASIA CCS ’24. Singapore, Singapore: Association for Computing Machinery, 2024, pp. 1600–1616. ISBN: 9798400704826. DOI: 10.1145/3634737.3644993. URL: https://doi.org/10.1145/3634737.3644993.

[46] libtom. *LibTomCrypt: A Modular and Portable Cryptographic Toolkit*. 2012. URL: https://github.com/libtom/libtomcrypt.

[47] Tein-Hsiang Lin and Wernhuar Tarng. “Scheduling periodic and aperiodic tasks in hard real-time computing systems”. In: *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS ’91. San Diego, California, USA: Association for Computing Machinery, 1991, pp. 31–38. ISBN: 0897913922. DOI: 10.1145/107971.107976. URL: https://doi.org/10.1145/107971.107976.

[48] Ming Ling, Xin Xu, Yushen Gu, and Zhihua Pan. “Does the ISA Really Matter? A Simulation Based Investigation”. In: *2019 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. 2019, pp. 1–6. DOI: 10.1109/PACRIM47961.2019.8985059.

[49] C. L. Liu and James W. Layland. “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”. In: *J. ACM* 20.1 (Jan. 1973), pp. 46–61. ISSN: 0004-5411. DOI: 10.1145/321738.321743. URL: https://doi.org/10.1145/321738.321743.

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

[51] Ramya Jayaram Masti, Claudio Marforio, Aanjhan Ranganathan, Aurélien Francillon, and Srdjan Capkun. “Enabling trusted scheduling in embedded systems”. In: *Proceedings of the 28th Annual Computer Security Applications Conference*. ACSAC ’12. Orlando, Florida, USA: Association for Computing Machinery, 2012, pp. 61–70. ISBN: 9781450313124. DOI: 10.1145/2420950.2420960. URL: https://doi.org/10.1145/2420950.2420960.

[52] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. “Flicker: an execution infrastructure for tcb minimization”. In: *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. Eurosys ’08. Glasgow,

Scotland UK: Association for Computing Machinery, 2008, pp. 315–328. ISBN: 9781605580135. DOI: 10.1145/1352592.1352625. URL: https://doi.org/10.1145/1352592.1352625.

[53] MITRE Corporation. *MITRE ATT&CK® Matrix for Industrial Control Systems (ICS)*. 2025. URL: https://attack.mitre.org/matrices/ics/.

[54] Antonio Joia Neto, Norrathep Rattanavipanon, and Ivan De Oliveira Nunes. " PEARTS: Provable Execution in Real-Time Embedded Systems ". In: *2025 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 3765–3782. DOI: 10.1109/SP61157.2025.00047. URL: https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00047.

[55] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. "Sancus 2.0: A Low-Cost Security Architecture for IoT Devices". In: *ACM Trans. Priv. Secur.* 20.3 (July 2017). ISSN: 2471-2566. DOI: 10.1145/3079763. URL: https://doi.org/10.1145/3079763.

[56] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. "APEX: A Verified Architecture for Proofs of Execution on Remote Devices under Full Software Compromise". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 771–788. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/nunes.

[57] Trevor Perrin and Moxie Marlinspike. *The Double Ratchet Algorithm*. Version Revision 1. 2016. URL: https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf.

[58] Sandro Pinto, Jorge Pereira, Tiago Gomes, Mongkol Ekpanyapong, and Adriano Tavares. "Towards a TrustZone-Assisted Hypervisor for Real-Time Embedded Systems". In: *IEEE Computer Architecture Letters* 16.2 (2017), pp. 158–161. DOI: 10.1109/LCA.2016.2617308.

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

[60] Sandro Pinto and Nuno Santos. "Demystifying Arm TrustZone: A Comprehensive Survey". In: *ACM Comput. Surv.* 51.6 (Jan. 2019). ISSN: 0360-0300. DOI: 10.1145/3291047. URL: https://doi.org/10.1145/3291047.

[61] Qualcomm Incorporated. *Qualcomm Trusted Execution Environment (QTEE)*. Version 80-70015-11 Rev. AB. Oct. 2024. URL: https://docs.qualcomm.com/bundle/publicresource/topics/80-70015-11/qualcomm-trusted-execution-environment.html.

[62] R. Rajkumar. “Real-time synchronization protocols for shared memory multiprocessors”. In: *Proceedings.,10th International Conference on Distributed Computing Systems*. 1990, pp. 116–123. DOI: 10.1109/ICDCS.1990.89257.

[63] Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky. “Real-time synchronization protocols for multiprocessors”. In: *Proceedings of the 9th IEEE International Real-Time Systems Symposium (RTSS)*. IEEE, 1988, pp. 259–269. DOI: 10.1109/REAL.1988.51115.

[64] Jonas Röckl, Christian Lindenmeier, Matti Schulze, and Tilo Müller. “Conditional Network Availability: Enhancing Connectivity Guarantees for TEE-Based Services”. In: *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 2024, pp. 225–233. DOI: 10.1109/EuroSPW61312.2024.00030.

[65] S. Rohith, S. K. Miruthula, Shreyas K, and Vaidehi Vijayakumar. “Performance Comparison of Multicore Architectures”. In: *2023 6th International Conference on Recent Trends in Advance Computing (ICRTAC)*. 2023, pp. 830–835. DOI: 10.1109/ICRTAC59277.2023.10480854.

[66] Selma Saidi, Rolf Ernst, Sascha Uhrig, Henrik Theiling, and Benoît Dupont de Dinechin. “The shift to multicores in real-time and safety-critical systems”. In: *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 2015, pp. 220–229. DOI: 10.1109/CODESISSS.2015.7331385.

[67] *Secure Hash Standard (SHS)*. Aug. 2015. URL: https://doi.org/10.6028/NIST.FIPS.180-4.

[68] L. Sha, R. Rajkumar, and J.P. Lehoczky. “Priority inheritance protocols: an approach to real-time synchronization”. In: *IEEE Transactions on Computers* 39.9 (1990), pp. 1175–1185. DOI: 10.1109/12.57058.

[69] Ron Stajnrod, Raz Ben Yehuda, and Nezer Jacob Zaidenberg. “Attacking TrustZone on devices lacking memory protection”. In: *Journal of Computer Virology and Hacking Techniques* 18.3 (2022), pp. 259–269. DOI: 10.1007/s11416-021-00413-y.

[70] J.K. Strosnider, J.P. Lehoczky, and Lui Sha. “The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments”. In: *IEEE Transactions on Computers* 44.1 (1995), pp. 73–91. DOI: 10.1109/12.368008.

[71] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. “OAT: Attesting Operation Integrity of Embedded Devices”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1433–1449. DOI: 10.1109/SP40000.2020.00042.

[72] Rapita Systems. *What Really Happened to the Software on the Mars Pathfinder Spacecraft*. July 2013. URL: https://www.rapitasystems.com/blog/what-really-happened-software-mars-pathfinder-spacecraft.

[73] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. “CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management”. In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1057–1074. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang.

[74] Daniel Terhell. *Windows and Real-Time*. 2014. URL: https://www.osr.com/nt-insider/2014-issue3/windows-real-time/.

[75] Trusted Computing Group. *Foundational Trust for IoT and Resource-Constrained Devices*. White Paper. Trusted Computing Group, June 2017. URL: https://trustedcomputinggroup.org/wp-content/uploads/Foundational-Trust-for-IOT-and-Resource-Constrained-Devices.pdf.

[76] Trusted Computing Group. *Hardware Requirements for a Device Identifier Composition Engine*. Specification. Version 1.0. Trusted Computing Group, Aug. 2024. URL: https://trustedcomputinggroup.org/wp-content/uploads/Hardware-Requirements-for-a-Device-Identifier-Composition-Engine-Version-1.0-Revision-0.91_pub.pdf.

[77] Trusted Computing Group. *TCG Attestation Framework, Part 1: Terminology, Concepts, and Requirements*. Version 1.0 RC 1 (Public Review). “Work in Progress” specification; released May 20, 2025. Trusted Computing Group, May 2025. URL: https://trustedcomputinggroup.org/wp-content/uploads/TCG-Attestation-Framework-Part-1-RC-1_20May2025.pdf.

[78] Trusted Computing Group. *TCG D-RTM Architecture*. Tech. rep. Version Version 1.0.0. Trusted Computing Group, June 2013. URL: https://trustedcomputinggroup.org/wp-content/uploads/TCG_D-RTM_Architecture_v1-0_Published_06172013.pdf.

[79] Trusted Computing Group. *Trusted Platform Module 2.0 Library Part 0: Introduction*. Technical Specification. Version Version 184. Published March 20, 2025. Trusted Computing Group, Mar. 2025. URL: https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-2.0-Library-Part-0-Version-184_pub.pdf.

[80] Trusted Computing Group. *What Is a Root-of-Trust (RoT)?* 2025. URL: https://trustedcomputinggroup.org/about/what-is-a-root-of-trust-rot/.

[81] *Trusted Firmware-A (reference implementation for secure world software)*. GitHub repository mirror of Trusted Firmware-A. Secure world firmware for Arm A-Profile architectures (Armv7-A and Armv8-A). 2025. URL: https://github.com/ARM-software/arm-trusted-firmware.

[82] TrustedFirmware.org. *OP-TEE Documentation*. 2025. URL: https://optee.readthedocs.io/en/4.5.0.

[83] Trustonic. *Kinibi-520a: The Latest Trustonic Trusted Execution Environment (TEE)*. 2021. URL: https://www.trustonic.com/technical-articles/kinibi-520a-the-latest-trusted-execution-environment-tee/.

[84] Tom Van Eyck, Sam Michiels, Xiaojiang Du, and Danny Hughes. "NetReach: Guaranteed Network Availability and Reachability to enable Resilient Networks for Embedded Systems". In: *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 2024, pp. 242–244. DOI: 10.1109/EuroSPW61312.2024.00032.

[85] Tom Van Eyck, Hamdi Trimech, Sam Michiels, Danny Hughes, Majid Salehi, Hassan Janjuaa, and Thanh-Liem Ta. "Mr-TEE: Practical Trusted Execution of Mixed-Criticality Code". In: *Proceedings of the 24th International Middleware Conference: Industrial Track*. Middleware '23. Bologna, Italy: Association for Computing Machinery, 2023, pp. 22–28. ISBN: 9798400704277. DOI: 10.1145/3626562.3626831. URL: https://doi.org/10.1145/3626562.3626831.

[86] Jinwen Wang, Ao Li, Haoran Li, Chenyang Lu, and Ning Zhang. "RT-TEE: Real-time System Availability for Cyber-physical Systems using ARM TrustZone". In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022, pp. 352–369. DOI: 10.1109/SP46214.2022.9833604.

[87] Jinwen Wang, Yujie Wang, Ao Li, Yang Xiao, Ruide Zhang, Wenjing Lou, Y. Thomas Hou, and Ning Zhang. "ARI: Attestation of Real-time Mission Execution Integrity". In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 2761–2778. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/wang-jinwen.

[88] Jin-yong Yin and Guo-chang Guo. "An Algorithm for Scheduling Aperiodic Real-Time Tasks on a Static Schedule". In: *2009 Second International Conference on Information and Computing Science*. Vol. 1. 2009, pp. 70–74. DOI: 10.1109/ICIC.2009.25.

[89] Dylan Zueck, Nathaniel Atallah, Ian Do, Zhihao Yao, and Ardalan Amiri Sani. "Hora: High Assurance Periodic Availability Guarantee for Life-Critical Applications on Smartphones". In: *Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems*. APSys '24. Kyoto, Japan: Association for Computing Machinery, 2024, pp. 115–121. ISBN: 9798400711053. DOI: 10.1145/3678015.3680486. URL: https://doi.org/10.1145/3678015.3680486.

Appendix A

Survey on development boards

For this study, we parsed the specification and datasheet of the SoCs, that of their security components, and analyzed platform-specific implementations in open-source projects such as U-Boot, TF-A, and OP-TEE when the available documentation did not provide enough detail. In total, we analyzed about 30 development kits featuring multicore Cortex-A profile SoCs. We omit the boards for which we could not find sufficient relevant documentation, and report our findings in Table A.1.

A board is reported to have a secure timer (*Sec. timer*, the value is ● for *yes*) if (1) it provides access to a secure processor timer, or (2) if external timers are available on the platform and it is possible to assign them to the Secure world. Should secure timers be present on the platform but not exposed to developers, a ◐ will be shown. The *ASC/PC* condition is met (●) when the board is equipped with TrustZone components to partition memory and peripherals into Secure and Non-Secure splits. A ○ indicates that the condition is not met. The same rule applies for the *GIC* column, where the condition is met if and only if an interrupt controller with security extension is available on the platform. The Unique Device Secret capabilities (*UDS cap.*) column groups the two requirements for the UDS, which are (1) secure storage, and (2) locking mechanism to restrict access to the device secret after the first DICE flow. A ○ indicates that either or both conditions are not met.

Secure physical processor timers are available on all surveyed boards. In fact, those timers are available on all Armv8 Cortex-A processors [86]. Although OP-TEE has been ported to Rpi boards, the implementation remains insecure [69], as they do not benefit from the necessary TrustZone hardware components such as an Address Space Controller or Peripheral Controller, therefore not enforcing strict segmentation of resources between the Normal and Secure worlds. Regarding the UDS capabilities, most of these platforms are equipped with a security coprocessor. Nvidia boards have the Jetson Security Engine (SE), Zynq UultraScale+ MPSoc-based boards have the Platform Management Unit (PMU), i.MX boards have the EdgeLock (ELE), TI boards have the Device Management and Security Control (DMSC), and STM32 platforms have (Boot and Security and OTP control) BSEC.

Board Name	SoC Name	Sec. timer	ASC/PC	GIC	UDS cap.
Raspberry Pi 3 B	BCM2837	●	○	●	○
Raspberry Pi 4 B	BCM2711	●	○	●	○
Raspberry Pi 5	BCM2712	●	○	●	○
Jetson TX2 DK	Nvidia Jetson TX2	●	●	●	○
Jetson AGX Xavier	Nvidia Xavier	●	●	●	●
Jetson Orin Nano	Nvidia Orin	●	●	●	●
Jetson Orin NX	Nvidia Orin NX	●	●	●	●
Jetson AGX Orin	Nvidia Orin AGX	●	●	●	●
Xilinx ZCU102	Zynq UltraScale+ MPSoC	●	●	●	●
Xilinx ZCU104	Zynq UltraScale+ MPSoC	●	●	●	●
MCIMX8ULP-EVK	NXP i.MX 8ULP	●	●	●	●
i.MX93EVK	NXP i.MX 93	●	●	●	●
i.MX95EVK	NXP i.MX 95	●	●	●	●
HiKey 960	HiSilicon Kirin 960	●	●	●	○
TMDX654IDKEVM	TI AM65x	●	●	●	●
TMDSAM64EVM	TI AM64x	●	●	●	●
Arm Juno r2	Juno SoC	●	●	●	○
STM32MP157F-EV1	STM32MP157F	●	●	●	●
STM32MP257F-EV1	STM32MP257F	●	●	●	●

Table A.1: Aggregated findings on the presence of the hardware components required to implement our design.

Appendix B

Turning legacy RT tasks into secure RT tasks

The following discussion focuses on control loops implemented in the C programming language, intended to be executed on Linux with the `PREEMPT_RT` patches, and how they can be transformed into critical real-time tasks for the Secure scheduler. It is not intended as a guide for developing generic Trusted Applications.

Listing B.1 shows the source code of a sample real-time application written for Linux. It is transformed into an equivalent Secure real-time task for our Secure scheduler and displayed in Listing B.2 is the result. The main changes are in how the control loop is scheduled. Other changes include the compulsory modification of the signature of the control loop routine and the necessary functions related to the TEE framework for TAs. The important detail to remember is that scheduling policies are defined in a separate file and provisioned to the Secure scheduler to initiate the loading of the real-time application. As a result, the configuration of the clock for scheduling is done transparently and the programmer only needs to use the `TEE_exit_job()` system call to signal the completion of the current job.

```
1  #define CONTROL_PRIO
2  #define CONTROL_PERIOD
3  #define CONTROL_CORE_AFFN
4
5  int sensor(void);
6  int compute(int value);
7  void actuate(int result);
8
9  void control_loop(void)
10 {
11     int value = sensor();
```

```c
    int result = compute(value);
    actuate(result);
}

int main(void)
{
    // Set RT policy and affinity
    set_realtime(pthread_self(), SCHED_FIFO, CONTROL_PRIO, CONTROL_CORE_AFFN);

    struct timespec next;
    clock_gettime(CLOCK_MONOTONIC, &next);

    // Wait until start of period
    clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &next, NULL);
    for (;;) {
        // Wait until start of period
        clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &next, NULL);

        control_loop();

         // Schedule next period
        timespec_add_us(&next, CONTROL_PERIOD);
    }

    // should not return
    return -1;
}
```

Listing B.1: Toy control loop written as a C program intended to run on Linux with PREEMPT_RT patches.

```c
#define TA_MAIN 0

int sensor(void);
int compute(int value);
void actuate(int result);

TEE_Result control_loop(void)
{
    int value = sensor();
    int result = compute(value);
    actuate(result);

    return TEE_SUCCESS;
}

void ta_main(void)
{
    TEE_Result rc;
```

```
19
20      // Policy is decoupled from the binary, no need to specify it in the code
21
22      for (;;) {
23          rc = control_loop();
24
25          // Syscall to return from a job
26          // The Secure scheduler will automatically reschedule at the next period
27          TEE_exit_job(rc);
28      }
29  }
30
31  /*
32   * Below are the TA lifecycle routines called by the framework.
33   */
34
35  TEE_Result TA_CreateEntryPoint(void)
36  {
37      return TEE_SUCCESS;
38  }
39
40  void TA_DestroyEntryPoint(void) {}
41
42  TEE_Result TA_OpenSessionEntryPoint(uint32_t ptype,
43                                      TEE_Param param[4],
44                                      void **session_id_ptr)
45  {
46      return TEE_SUCCESS;
47  }
48
49  void TA_CloseSessionEntryPoint(void *sess_ptr) {}
50
51  TEE_Result TA_InvokeCommandEntryPoint(void *session_id,
52                                        uint32_t command_id,
53                                        uint32_t parameters_type,
54                                        TEE_Param parameters[4])
55  {
56      switch (command_id) {
57      case TA_MAIN:
58          ta_main();
59
60          // should not return
61          return TEE_ERROR_GENERIC;
62      default:
63          return TEE_ERROR_GENERIC;
64      }
65  }
```

Listing B.2: The equivalent control loop written as a Trusted Application for the Secure scheduler.