17th International Conference on Space Operations, Dubai, United Arab Emirates, 6 - 10 March 2023.

SpaceOps-2023, ID # 459

Hardware-Based Isolation for Advanced Safety and Security in Spacecraft

David Koisser[ac]*, Ferdinand Brasser[b], Patrick Jauernig[b], Emmanuel Stapf[b], Marcus Wallum[c], Daniel Fischer[c], Ahmad-Reza Sadeghi[a]

[a] *System Security Lab, Technical University of Darmstadt, Germany,*
david.koisser@trust.tu-darmstadt.de, ahmad.sadeghi@trust.tu-darmstadt.de
[b] *SANCTUARY Systems GmbH,*
ferdinand.brasser@sanctuary.dev, patrick.jauernig@sanctuary.dev, emmanuel.stapf@sanctuary.dev
[c] *Applications and Robotics Data Systems Section, European Space Operations Centre, European Space Agency,*
marcus.wallum@esa.int, daniel.fischer@esa.int
* Corresponding Author

Abstract

With the advent of new mission concepts, such as multi-tenant spacecraft, interconnected spacecraft networks, or AI-supported autonomy, onboard spacecraft software needs to provide a growing number of functionalities. However, as onboard software grows more complex, the probability of software bugs rises as well, becoming an increasingly important factor in spacecraft safety, reliability, and cybersecurity considerations. In this paper, we introduce a novel software architecture for onboard software that builds on a strong hardware-assisted isolation mechanism. Our architecture leverages hardware extensions from Arm processors already deployed today (e.g., in CubeSats) that are becoming common in the space sector. By separating software components into hardware-assisted compartments, we ensure that they cannot affect each other, even when one component crashes. Further, our architecture allows to detect faulty software components and restart them into a safe configuration, reducing dependency on the spacecraft's safe mode. Especially for missions in which different parties jointly utilize (parts of) a spacecraft, such as hosted payloads or multi-tenant spacecraft, our architecture provides strong safety and cybersecurity guarantees due to the strong separation between components. Due to these properties operating spacecraft becomes inherently more reliable while simplifying onboard software development, as the inherent safety and cybersecurity guarantees reduce the need to extensively test individual software components or auditing of external software. We evaluated our novel software architecture thoroughly on a hardware development board.
Keywords: (maximum 6 keywords)

Acronyms/Abbreviations

ADCS	Attitude Determination and Control System
COTS	Commercial Off-The-Shelf
EL	Exception Level
HSM	Hardware Security Module
IMA	Integrated Modular Avionics
MMU	Memory Management Unit
OS	Operating System
ROM	Read-Only Memory
RTOS	Real-Time Operating System
SoC	System on a Chip
TA	Trusted Application
TEE	Trusted Execution Environment
TPM	Trusted Platform Module
TPS	Time and Space Partitioning
VM	Virtual Machine

1. Introduction

The trend of increasing the computational power of spacecraft enables novel use cases, e.g., satellite-as-a-service, where satellites are shared among various parties, as demonstrated by ESA's OPS-SAT mission [1]. This multi-tenant approach promises easy and cost-efficient access to satellites, paving the way for innovative mission concepts and collaborations among a plethora of players. Another relevant trend is the advent of large constellations of cheap spacecraft, which will likely require streamlining operations to scale to many spacecraft and, for example, incorporate AI-supported spacecraft autonomy. These new concepts offer tremendous advantages, yet, they also lead to a growing complexity of onboard software, and thus, the probability of software bugs rises. Software faults endanger not only the safety of future spacecraft but also their cybersecurity, as software faults often manifest as exploitable vulnerabilities. For example, both cybersecurity and safety might be compromised by faulty or malicious third-party software components onboard. To address these emerging challenges, new concepts are needed to make spacecraft ready for the requirements of the future.

A key concept for designing spaceflight software is the separation of different software components to prevent them from affecting each other. Strong separation of software components is a mature topic in the aerospace field, including avionics standards defining how onboard systems shall be designed [2], which also caught the interest of the space community [3,4]. Furthermore, virtualization techniques for stronger isolation of onboard software components have been a topic of interest in recent years. There have been approaches focusing on virtualization on top of an underlying operating system (OS) [5,6]. Nevertheless, the advent of hardware-assisted virtualization features on most modern chip architectures promises strong isolation guarantees without sacrificing performance. As the Arm architecture gradually finds more use in the space sector, such as Boeing's High-Performance Spaceflight Computer [7] or numerous CubeSat platforms, recent works have proposed to leverage the architecture's native virtualization capabilities for onboard software [8,9,10]. However, all these approaches come with their shortcomings. While virtualization techniques running on top of an OS can generalize the development of safety-increasing isolation between software components, they are still vulnerable to faults affecting the underlying OS and cybersecurity vulnerabilities. Approaches leveraging hardware-based virtualization improve on this, yet they are missing proper parallelism support for multi-core processors. We discuss the related work extensively in Section 7.

In this paper, we introduce a novel software architecture targeting spacecraft that exploits Arm-based Systems-on-Chips (SoCs), which can already be found in many CubeSat platforms today [11,12,13]. Our architecture combines virtualization technologies and the Arm TrustZone security primitives. The latter are hardware security features found on virtually all Arm chips released in the last decade, which allows for the creation of a Trusted Execution Environment (TEE). TEEs create strongly isolated execution compartments to protect sensitive software components and services from faulty or potentially malicious applications or system software co-located on the same computing platform. Our architecture combines these mechanisms with hardware virtualization techniques to isolate software components. This results in three key features for future spacecraft: (1) resiliency to tolerate faulty software components onboard, (2) lightweight and safe incorporation of multi-tenant mission concepts, and (3) structural improvement of cybersecurity for the onboard software execution. These features are achieved while providing multi-core support allowing the assignment of cores to specific VMs, and safe interrupt handling with the exclusive assignment of devices per VM. Both these features allow for better real-time guarantees while simplifying software development. Due to this low-level safety and security concept based on VMs, the effort to adapt existing software to our architecture is low. To evaluate our architecture, we implemented and benchmarked a proof-of-concept for a CubeSat and deployed it on a representative Arm-based platform with the required virtualization and TrustZone extensions.

2. Background

In this section, we provide background on modern hardware and cybersecurity features our architecture leverages.

2.1. Arm TrustZone

The TrustZone security architecture first announced in 2004 for Arm-based for Systems-on-Chip (SoC) comprises a set of security extensions and features to enable the protected processing of sensitive code and data. It covers the processor, the memory subsystem (including the memory controller and caches), and the bus systems connecting peripheral devices—both integrated in the SoC and external peripheral devices.

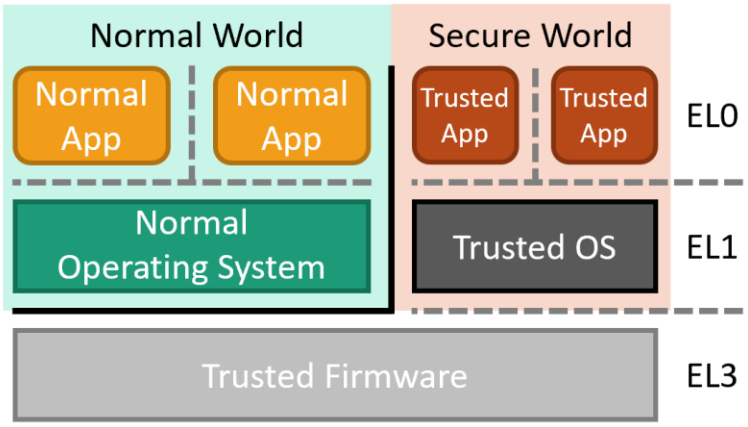

Figure 1: Overview of Arm TrustZone's separation of normal and secure world.

Figure 1 shows an overview of how TrustZone works. Here, a system can be partitioned into two separate worlds: A secure world and a normal world. The secure world is strongly isolated from the normal world, which prevents code executed in the normal world from manipulating or interfering with code or data included in the secure world. The secure world hosts security-critical data and services, while the normal world hosts the bulk of the system's software, including legacy code. This is achieved by extending the execution state of TrustZone-enabled processors with status information whether the processor is operating in a secure state (i.e., in the secure world) or in a non-secure state (i.e., in the normal world), which is orthogonal to the privilege levels for code execution (Exception Levels—EL0-EL3, which are used to execute applications with fewer privileges than the operating system). This execution state is maintained independently for each processor in a multi-core system. The security status information is further propagated within the SoC, so access to peripheral devices or individual memory sections can be controlled based on the security state of the processor and the code attempting to access them.

In a typical TrustZone-enabled system, a portion of the main memory is reserved exclusively for the secure world, and peripheral devices (e.g., a cryptographic accelerator) are exclusively reserved for the secure world. In the normal world, legacy software runs a standard operating system and various applications. In the secure world, a dedicated operating system, called a trusted OS, manages a number of so-called trusted applications (TAs) that perform security services. Normal-world software can benefit from these security services through a defined interface (e.g., following the FFA standard [14]) that allows data exchange and remote-function calls between the software in both worlds. The limitation of TrustZone lies in the fact that exactly two worlds exist on a system, and thus, all security-critical code has to share the secure world. In a multi-tenant scenario, where multiple parties want to protect their own code and data independently of all other parties, the TrustZone architecture is insufficient.

2.2. Virtualization

Virtualization allows the parallel execution of multiple virtual machines (VMs) on a single hardware machine, where each VM has the impression of operating on its own hardware machine. Maintaining multiple VMs on a single hardware machine is done by a software component called *hypervisor*. The hypervisor is responsible for managing the available hardware resources, like processor execution time, memory, peripheral devices, and maintaining the state of all VMs. In particular, the hypervisor has to intercept all interactions of software running in a VM with the system's hardware to prevent one VM's actions from impacting any other VM. For instance, a VM must not be allowed to issue a reboot of the platform; instead the hypervisor has to reset the state of the issuing VM such that the software within it reboots while all other VMs continue to execute.

Management of the core resources includes scheduling VMs on the system's CPU cores and partitioning the system memory, i.e., reserving exclusive memory regions for each VM. Many modern processor architectures provide hardware features supporting the hypervisor, such as a dedicated privileged level for the hypervisor (e.g., EL2 in Figure 1), additional hypervisor-exclusive timers, 2nd-level memory translation, and virtual interrupts systems. Leveraging these features, a hypervisor can run with higher privileges than the OSes of each VM and perform time-based scheduling using timers not visible to the VMs. Further, the hypervisor can perform memory translation and access control such that each VM has the impression of having access to physical memory starting at address zero. Additionally, the hypervisor can intercept interrupts in the system to translate and redistribute them to the correct VMs. The hardware support enables all these hypervisor operations to induce minimal performance overhead.

2.3. Modern Cybersecurity Features

In the last one-and-a-half decades, numerous security architectures were proposed, which base their security guarantees on hardware features. Arm TrustZone and virtualization present only two of these hardware features. In the following, we shortly introduce the most important cybersecurity features for our architecture, namely secure boot and attestation.

The purpose of secure boot is to verify the integrity of a system's software while it is booting. This is achieved by forming a chain of trust among all software components on the system in which one component always verifies the integrity of the next element in the chain. Only if the successor component was proven to be unmodified, execution control is passed on to it. The chain of trust starts with the Root of Trust (RoT), e.g., BIOS or UEFI, typically stored in a specific memory region that can only be written once, e.g., in Read-only Memory (ROM). The last element in the chain of trust is typically the operating system's kernel. After execution control is given to the kernel, the boot process is finished. The verification at each stage of the boot process is performed by computing a cryptographic hash digest of the next element's binary and comparing it to a reference integrity measurement. The most common approach to store the measurements is a digital certificate signed by a signing authority, e.g., the system vendor, and to include them in the software components. During secure boot, the verifying component can then verify the integrity of the reference measurements using the public verification key of the signing authority.

In many scenarios, the integrity of the system's software should also be verifiable by external entities to have a guarantee which software was booted on the system. Here, one entity, called the *prover*, proves its (software) state to another entity, called the *verifier*. This is typically referred to as *attestation*. The secure boot feature provides a good basis for verifying the state of a system to an external party since it already provides a set of verified hash digests representing the software stack booted on the system. Thus, the prover may sign its state measurement along with a unique nonce and send it to the verifier.

3. Design

This section describes the design of our novel onboard software architecture. For this, we first define the goals of our architecture. Afterward, we present the architecture in detail and how our approach meets the defined goals. Finally, we show an example onboard software setup to illustrate how our architecture can enhance spaceflight scenarios.

3.1. Design Goals

The goal of our design is to provide the following unique set of features and properties for multi-tenant spacecraft:

- **G1**: *Fault detection and recovery*: The architecture shall enable the detection of faults in individual software components; furthermore, faulty components must be brought to a correct state to recover them.
- **G2**: *Secure operation and maintenance*: The architecture shall enable its components to process, store, and securely transmit sensitive information. Maintenance operations, like control commands or updates, must be handled securely as well.
- **G3**: *Standard-compliant security services*: The architecture shall provide security services to all software following standardized APIs to enable secure-by-default software designs.
- **G4**: *Multi-tenant operation*: The architecture shall support the safe and secure parallel execution of multiple independent workloads (e.g., experiments), allowing such workloads the access and use of peripherals (e.g., sensors).
- **G5**: *Communication and collaboration*: The architecture shall allow platform components to securely and efficiently communicate with each other to enable collaboration and information sharing.
- **G6**: *Legacy software support*: Existing software shall be executable with only non-invasive modification and adaption efforts.
- **G7**: *Real-time guarantees*: For critical workloads and base functionalities, the architecture shall be able to provide real-time execution guarantees.

3.2. Design Details

The core of our architecture is its system design, which is fundamental for maintaining integrity, confidentiality, and isolation throughout the entire lifecycle of the spacecraft's onboard software. Figure 2 shows our architecture with its core components, i.e., a *custom-build hypervisor* that manages the platform's resources and a *security agent* that provides the functionalities to bootstrap and maintain the platform's integrity. Our novel architecture integrates both, allowing them to tightly collaborate in order to achieve all goals listed above.

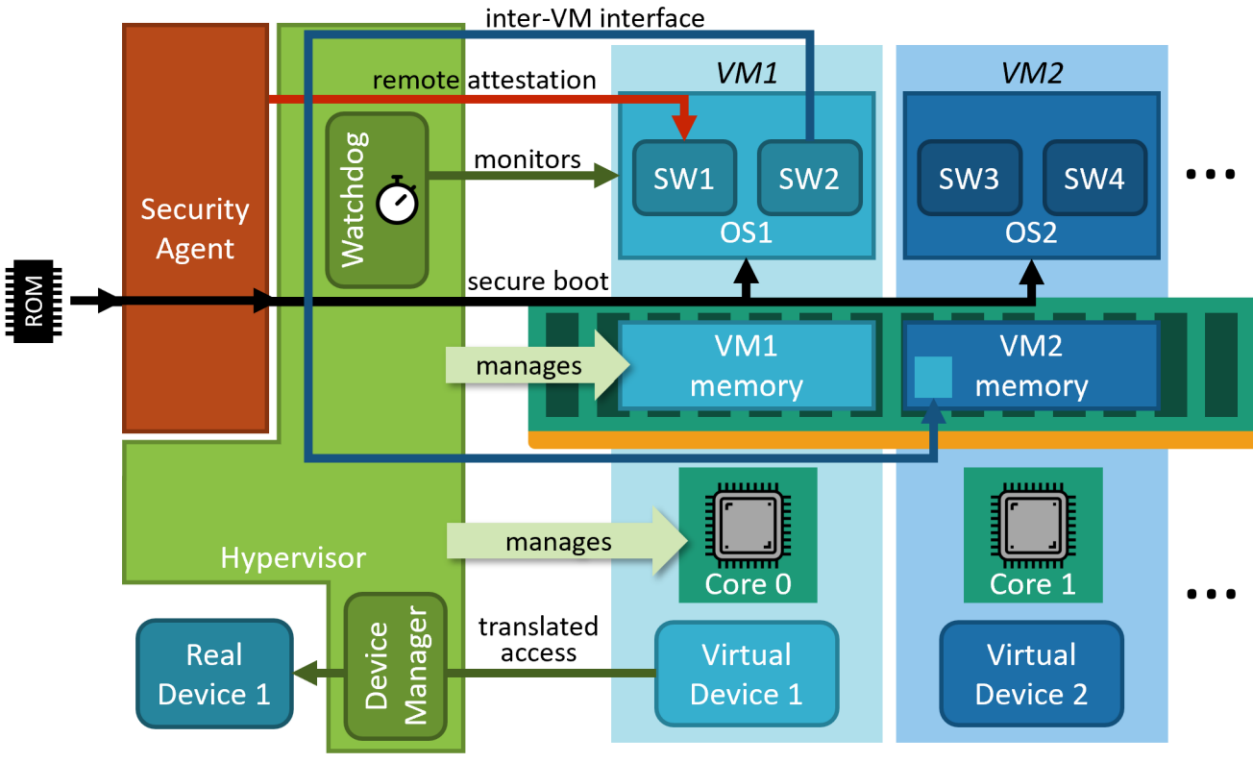

Figure 2: Overview of components and their interactions in our architecture.

On the one hand, the hypervisor is responsible for assigning the platform's hardware resources to individual VMs, including processing units (CPUs), memory regions, and peripheral devices. These assignments are done in a way that isolation between VMs is ensured on the hardware level via strict resource partitioning, e.g., memory or CPU cores, or via explicit access control mechanisms. With the help of the security agent, the hypervisor can also establish interfaces between the VMs to allow software running in different VMs to collaborate. Furthermore, the hypervisor incorporates a hardware-based *watchdog*, which monitors the VMs to detect faults (e.g., crashes) and can restart them for recovery.

On the other hand, the security agent ensures the integrity of the entire platform's software stack, including the security agent and hypervisor themselves via *secure boot*. Another integral responsibility of the security agent is to assign secure identities to the VMs. Every VM identity is unique and allows remotely verifying a VM's setup and configuration via *remote attestation*, e.g., by another tenant. In addition, every VM has associated permissions to access peripherals and cryptographic accelerators. Moreover, the security agent is responsible for enforcing identity-based permissions on the communication channels the hypervisor opens, and guarantees authenticity, confidentiality, and integrity protection for them. In the following, we will describe the individual components of our architecture in detail.

3.2.1. Security Agent

The security agent runs in the secure world of Arm TrustZone (see Section 2.1.1) and ensures the integrity of all components, as well as the dynamic configuration of the platform, from the first step of the boot process leveraging *secure boot*. Starting from code stored in read-only memory (ROM), which is therefore inherently integrity-protected, all components and their configurations are verified to be unmodified during startup. For each component, including the security agent, the hypervisor, and the individual VMs' OSes, a cryptographic hash of their code and data is calculated and compared against a reference value. Only if the reference value match, i.e., the component has not been modified, will it be allowed to execute. The entire process is logged, allowing us to securely identify all components running on the system using cryptographic hash digests calculated for each in the boot process. Throughout the system's run time, these digests are associated with the unique identity of each VM; the system's configuration, i.e., the resource allocation, assignment of peripheral devices, and the permission to interact with other VMs or the security agent, is specified using the VM's identities.

Furthermore, using secure identities for each VM enables the security agent to provide services to all VMs, including the provisioning of individual cryptographic keys for each VM to encrypt sensitive information and enable security services like remote attestation (see Section 2.2.1). Additionally, the security agent offers *security services following industry standards* (**G3**), such as TPM 2.0 compliant [15] virtual TPMs for each VM as well as virtual hardware security modules (vHSM) through the PKCS#11 API [16], allowing every VMs to use an isolated instance of a TPM or HSM without having to fear interference from other VMs. Further virtualization of these components eliminates the costs for dedicated physical chips and saves weight.

The integrated identity management of our design enables the *secure management and update of individual components and configurations of a system* (**G2**). New components are verified based on their identity, i.e., their code's hash, before installation and subsequently during every system start via secure boot. Thus, component-based identity management allows the update of individual components by loading a new version of a component via the network onto the system and verifying the authenticity of that update via the new component's identity. As VMs are individual and independent components, they can be updated individually, even at run time, by restarting the one VM that needs updating.

3.2.2. Hypervisor

The hypervisor ensures the isolation of VMs at run time by providing isolated environments for each VM such that they can operate independently. This allows to *securely execute workloads—running in individual VMs—from different vendors on the same platform* (**G4**). For this, a section of the main memory is allocated for each VM. Each VM's memory section is virtualized using a 2nd level memory management unit (MMU), which translates all memory accesses of a VM such that all memory accesses of a VM will be directed to a memory location that the VM has been assigned. The memory access of each VM is translated based on a set of page tables managed by the hypervisor (independently of the memory translation performed based on the page tables managed by the VM's OS). Through this mechanism, each VM can access its assigned memory and only its assigned memory, ensuring the desired isolation between VMs. Furthermore, each VM can access its memory starting from address zero, i.e., as if it were the only software running on the system. As a result, each VM has the impression of having full access to the entire memory of its host system and does not have to consider the memory of other VMs, which enables the *execution of legacy workloads with minimal adaption efforts* (**G6**) on top of the hypervisor.

The execution units of the platform (i.e., CPU cores) are also assigned to the individual VMs by the hypervisor. Workloads with strict requirements in terms of execution and response times get assigned to a dedicated CPU core[*] if required. For less critical workloads, CPU cores can be shared using a time-sliced scheduling scheme, i.e., each VM gets to execute for a certain amount of time, then gets interrupted, the hypervisor stores its state, chooses the next VM for execution, restores the new VM's state, and proceeds where it was last interrupted. For scheduling VMs, the hypervisor leverages a dedicated timer, which is only available to the hypervisor, allowing each VM's OS to use their internal timers for their scheduling of applications[†].

Furthermore, the hypervisor handles all interactions the VMs have with the platform, including *device emulation* and *interrupt handling*. For interrupt handling, the hypervisor receives all interrupts (pausing the VM's execution) and forwards them to the respective VM. Since VMs are interrupted, the handling of interrupts by the hypervisor must be bound in execution time to be able to provide real-time execution guarantees for critical workloads. To prevent interrupts originating in the actions of one VM from impacting another, only interrupts specific to a critical VM are handled by the CPU core assigned to that VM. This is achieved by configuring the interrupt routing functionality of the platform's hardware interrupt control. The independence of VM's interrupts and the bounded execution time for interrupt handling by the hypervisor provides critical VMs the means to *perform real-time operations* (**G7**), as they can rely on the fact that they can execute without being interrupted at a predictable point in time or for an unpredictable duration.

Additionally, the hypervisor provides a watchdog component allowing VMs to set up hardware-based timers, which the VM's software has to reset at regular intervals. If a VM encounters a fault leading to a crash or halt of the VM's software, it cannot reset the watchdog timer, in which case the hypervisor receives an interrupt. This allows the hypervisor to *detect a failed VM and recover it* (**G1**), e.g., by restarting the VM, in which case it will be recovered to a known state that has been verified not to have been manipulated during the secure boot verification process. Alternatively, this mechanism can be extended to capture explicit state snapshots representing safe states to which a VM can be reset in case of a fault.

[*] Multiple CPU cores can be assigned as well.

[†] Rescheduling of VMs is not limited to the expiration of a time slice, other events like interrupts from peripheral devices can also lead to a rescheduling.

Furthermore, the hypervisor provides a mechanism to establish communication channels between VMs. Communication is only allowed between VMs within the access-control list bound to the corresponding identities. Communication partners can rely on the hypervisor to only send and receive data from the intended VM, *enabling secure collaboration* (**G5**) between workloads. The communication channels allow the workloads to exchange data with one another and send and receive signals from communication partners via a ring buffer each VM maintains in its own memory section to receive data. Signals are delivered to VMs by letting the hypervisor inject virtual interrupts, which a VM can freely decide to mask or disable at any point in time.

The hypervisor can assign peripheral devices exclusively to one VM, managing access control either with an IOMMU (In-/Output Memory Management Unit) or a simpler DMA access management component. Here, a VM's access to device memory of individual devices is limited, i.e., the memory addresses to which the configuration registers and data buffers of the devices are remapped. For devices that are shared between multiple VMs, we leverage an approach known as *para-virtualization*, where an abstract, simplified device (representing an entire class of devices, e.g., a block device) is emulated by a dedicated device driver VM. A VM can then access the device through a so-called virtio driver [17] for the device, which is an open para-virtualization standard and is prevalent in mainstream OSes. In our architecture, a dedicated driver VM is created that gets the device assigned. This dedicated VM exposes the standardized virtio-interface to other VMs for para-virtualized device access. Other VMs can send data to and receive data from the driver VM, which in turn multiplexes access to the physical device.

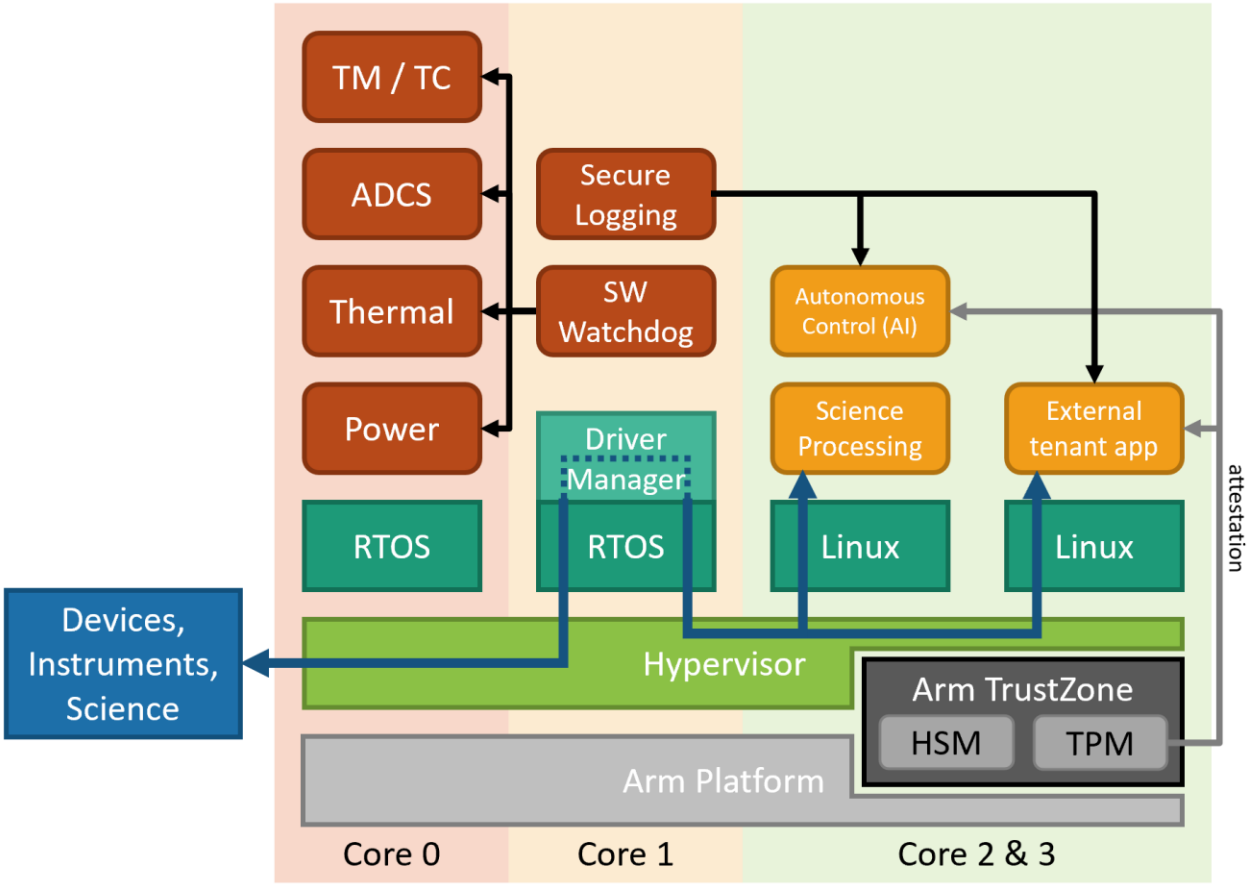

Figure 3: Exemplary spacecraft onboard software setup of our architecture.

3.3. Example Spacecraft Architecture

Implementing the design presented in the previous section leads to a highly reliable and secure spacecraft architecture. Figure 3 shows an example onboard software setup on an SoC with four cores. The first core is exclusively assigned to a workload running a Real-Time Operating System (RTOS), which in turn hosts the most critical software components to maintain the spacecraft, such as Attitude Determination and Control System (ADCS). Another RTOS workload is assigned to the second core, hosting important but less critical software components. This includes driver management to handle device access, a secure logging component, and a software-based

watchdog that monitors other applications and services. The other two cores are shared by two Linux VMs, one for internal science as well as AI processing, and one for other tenants' deployed software.

The exclusive use of cores for each RTOS workload inherently provides strong isolation guarantees, as any problems in other software components, especially for the Linux applications, cannot affect the more critical systems. Separating the critical and the important components into two RTOSes further improves onboard stability. Even one OS crashing due to a fatal fault, such as a memory violation, does not affect any other workloads' components. The inherent isolation and exclusive resource assignment also reduce the complexity of any real-time guarantees, as scheduling CPU time is not an issue, and interrupts are handled by our hypervisor. Further, this can be extended for the autonomous operation of the spacecraft. If an AI software component controls the spacecraft, the effect of any faulty decisions or bugs can be limited by cutting the component's access to, e.g., the ADCSes. Being able to deploy multiple VMs with separate workloads, along with the inherent real-time guarantees, greatly simplifies the adaption and deployment of legacy software. For instance, an old application could just be deployed to an environment it was designed for, the entire environment deployed as a VM to the spacecraft, and the configured core assignment ensures there is no interference.

The isolated VM approach further allows the deployment of other tenants' software without a lengthy, costly, and error-prone pre-validation process, as the inherent isolation protects other components from being affected by third-party bugs. Moreover, if the tenant deploys different software (or update) than announced, the attestation process will simply reject the uploaded software, preventing any malicious deployment. Opposed to the traditional approach of working with checksums in software, remote attestation uses collision-resistant hash functions calculated by the TPM and thus provides strong cybersecurity guarantees. The low-level assignment of devices to specific components prevents, e.g., a tenant's unwarranted blocking of a device that is needed by the internal science processing component. Using a secure logging component in an isolated VM also ensures there is a trace of a faulty or malicious tenant's software to establish accountability. As the HSM functionality is virtualized as well, even two different tenant's software can, e.g., securely store encryption keys without trusting the OS.

4. Evaluation

In this section, we evaluate the performance characteristics of our novel onboard software architecture. We implemented a prototype of our software architecture on a Toradex Verdin i.MX 8QM with 4GB of RAM and a Linux 5.15 with a buildroot environment as a VM. In the following, we measure the performance of individual steps and service invocations (micro benchmarks), macro benchmarks, and a spaceflight-specific use case.

4.1. Micro Benchmarks

We first evaluate single steps and service invocations to give an overview of the performance impact of our software architecture. The results are presented in Table 1. Compared to the VM boot, the time to initialize the hypervisor and the security agent is negligible, especially considering that the verification of the VMs, i.e., checking integrity and signature, is also done at the security agent's initialization. At run time, we measured the mean time for creating a remote attestation report for an entire VM over 100 runs and averaged the results. Note that this takes significantly longer, as the VMs contents have to be hashed and signed.

Table 1. Time measurements for micro benchmarks on the native platform and with our software architecture. Remote attestation has been averaged over 100 runs, times are in seconds.

Step		**Time [s]**
Boot		
	Security Agent initialization	0.339
	Hypervisor initialization	0.557
	Linux VM fully booted	7.610
Security Service Invocation		
	Remote Attestation	0.566

4.2. Macro Benchmarks

After evaluating individual steps and services of the architecture, we now want to analyze performance characteristics for real-world algorithms. For this, we use the rv8 benchmark suite [18], as it is self-contained, and therefore easy to cross-compile. We pinned all experiments to a single core within a VM and repeated all experiments over 100 runs. The benchmark results are depicted in Table 2, where we report the mean values for the individual benchmarks and an overall geometric mean. As can be seen, all experiments show (almost) identical run

times well within the magnitude of measurement errors (less than 0.04s). The reason for that is the fixed and isolated core assignment to VMs, and the VMs not triggering any interaction with the security agent.

Table 2: Time measurements for the rv8 benchmark on the native platform and with our software architecture. Averaged over 100 runs, times are in seconds.

Benchmark	Native [s]	Our Architecture [s]
aes	1.138	1.139
bigint	1.901	1.901
dhrystone	0.597	0.597
miniz	4.779	4.807
norx	0.532	0.532
primes	10.388	10.348
qsort	1.825	1.838
sha512	0.775	0.776
Geometric Mean		+0.141%

4.3. Spaceflight Software Evaluation

The previous experiment showcased performance characteristics for individual steps and services and macro benchmarks that perform more complex computations. To demonstrate practical performance overhead for spaceflight scenarios, we also conducted an experiment based on NASA's core Flight System (cFS) [19], a software framework for platform-independent mission software deployment. We inserted time measurement points into the cFS and compiled it for the ARM64 target architecture. Starting up cFS natively takes 1.5084s on average, while the startup on our onboard software architecture takes 1.5088s, measured over 100 runs. The time difference measured, especially the geometric mean of 0.141%, is clearly negligible in practice; hence, we deem our architecture practical.

5. Related Work

Isolating software components for aerospace onboard software is a long-established topic. The avionics industry has long since defined this under the term time and space partitioning (TSP) and standardized the general structure of RTOS in the Integrated Modular Avionics (IMA) architecture [2]. Briefly, this concept allows the execution of multiple applications on the same shared hardware while providing safety via strong isolation. Opposed to the previous federated architecture, which replicates the entire system, including the underlying hardware, IMA promises to require less power, weight, and cost. The space sector also seeks to adopt this concept [3,4]. Aside from standardized approaches, operational in-orbit software already uses similar principles of using partitioned software components. One example is NASA's core Flight System (cFS), which isolates functionality in separate apps that are monitored and can be independently restarted [19]. Another example is ESA's NanoSat MO Framework (NMF) [20], which uses segregated java components to partition its flight software for small spacecraft. All these approaches have a crucial aspect in common: The partitioning of the software components happens above the OS layer in the user space. This requires careful design of the core software components, such that they do not affect each other when faulty or, worse, the OS itself, which would bring the entire system down. Furthermore, this type of design is vulnerable to cyberattacks. A compromised software component may affect the OS, thus compromising the entire system.

To avoid the need for careful development of the onboard software to ensure isolation and safety, another approach is virtualization. Here, the software component runs in a virtual machine (VM) with its own virtual resources managed by the virtualization software. This allows leaving the mentioned development challenges mostly on the virtualization layer, independent of the onboard software components. One work in 2012 implemented a type 2 hypervisor (i.e., on top of a running OS) for a LEON4 processor on top of an RTEMS RTOS [5]. The authors argue that this increases fault tolerance as well as reusability, and they focus on reducing the virtualization overhead of their hypervisor. A different approach was presented in 2015, focusing on programming languages that come along with inherent isolation [6]. Mentioned examples of such languages are Python and Java, which primarily offer memory isolation, as opposed to C/C++. Memory violations are a common source of faults or security issues and can be avoided with this approach. The authors designed an onboard framework based on Lua, a lightweight scripting language often used for game development. The used Lua is also modified to, e.g., support real-time scheduling. However, these techniques are still running on top of an OS and rely on the safety guarantees of the underlying implementation.

As outlined in Section 2, many new hardware features have found their way to COTS processors and SoCs in recent years. Most notable for onboard software is the virtualization feature that allows running virtual OS' with

practically native performance while providing isolation on the hardware layer. Naturally, these features have sparked an interest in being used for onboard spacecraft software. In 2016, a hypervisor based on Arm TrustZone was proposed to implement a low-level layer providing time and space partitioning [8]. Thus, the proposed RTZVisor is a type 1 hypervisor working on bare metal, i.e., directly on the hardware without requiring an additional OS. It includes a cyclic scheduler for real-time guarantees and a health monitor for restarting guest VMs. This design was enhanced in 2017 with better performance and interrupt isolation for VM-exclusive use of devices [9]. Another approach was presented in 2018, which implements a Xen hypervisor to increase resilience against radiation faults via replication on a single hardware [10]. Here, the same VM runs multiple times, and all these VMs majority vote on the correct results. A crucial shortcoming of all approaches above is the lack of parallelism support, as only one VM (or its replicas) can run simultaneously.

In contrast, our scheme allows parallel execution of VMs on multiple cores, including exclusively assigning cores to specific VMs for better real-time guarantees without needing a low-level scheduler. The real-time guarantees are also enhanced with interrupt handling on the hypervisor level, preventing VMs from blocking others with interrupts. Enabling the execution of multiple isolated VMs also simplifies adapting legacy software to our architecture. Advanced monitoring and introspection capabilities allow for more extensive fault detection. Further, the transparent virtualization simplifies deploying legacy code to our new architecture. In terms of security, our architecture offers secure I/O access to devices, including exclusive use and policy-based management of devices. Additionally, we provide attestation services that allow us to validate running code among VMs securely, for software updates and even remotely attest code, e.g., for deployed third-party tenant software components.

6. Conclusion

With the emergence of trends like multi-tenant spacecraft, spacecraft networks, and AI, the increasing complexity of onboard software demands new approaches to ensure safety and security. Thus, we presented our novel software architecture to tackle these challenges. Our architecture defines a real-time capable hypervisor and a security agent, which combined ensure the platform's integrity and isolation while requiring only minimal adaption efforts for legacy software. To demonstrate the feasibility of our architecture, we implemented a prototype and evaluated it with a benchmark suite and NASA's open source cFS.

References

[1] ESA, OPS-SAT. https://www.esa.int/Enabling_Support/Operations/OPS-SAT, (accessed 20.01.23).
[2] ARINC, 653P0-3 Avionics Application Software Standard Interface, Part 0, Overview of ARINC 653. 17 November 2021, https://aviation-ia.sae-itc.com/standards/arinc653p0-3-653p0-3-avionics-application-software-standard-interface-part-0-overview-arinc-653, (accessed 20.01.23).
[3] J. Windsor, K. Hjortnaes, Time and space partitioning in spacecraft avionics, Third IEEE International Conference on Space Mission Challenges for Information Technology. IEEE, 2009. S. 13-20.
[4] ESA's Nebula Public Library, Development of partitioned Prototype Application (IMA-SP application development and software maintenance). 17 November 2022, https://nebula.esa.int/content/development-partitioned-prototype-application-ima-sp-application-development-and-software, (accessed 20.01.23).
[5] H. Joe, H. Jeong, Y. Yoon, H. Kim, S. Han, H.W. Jin, Full virtualizing micro hypervisor for spacecraft flight computer. In: 2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC). IEEE, 2012. S. 6C5-1-6C5-9.
[6] S. Park, H. Kim, S.Y. Kang, C.H. Koo, H. Joe, Lua-based virtual machine platform for spacecraft on-board control software. In: 2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing. IEEE, 2015. S. 44-51.
[7] ARM Community, New Arm-based computing system to enable deep space missions. 6 March 2018, https://community.arm.com/arm-research/b/articles/posts/new-arm-based-computing-system-to-enable-deep-space-missions, (accessed 20.01.23).
[8] S. Pinto, A. Tavares, S. Montenegro, Space and time partitioning with hardware support for space applications. Data Systems In Aerospace (DASIA), European Space Agency (Special Publication) ESA SP, 2016.
[9] S. Pinto, J. Martins, J. Lopes, M. Abreu, A. Tavares, SECSSY Hypervisor: security-safety synergy for aerospace, Data Systems in Aerospace, European Space Agency, Gothenburg, Sweden, 2017.
[10] D. Sabogal, A.D. George, Towards resilient spaceflight systems with virtualization. In: 2018 IEEE Aerospace Conference. IEEE, 2018. S. 1-8.
[11] EnduroSat, Onboard Computer (OBC). https://www.endurosat.com/cubesat-store/cubesat-obc/onboard-computer-obc/, (accessed 20.01.23).

[12] ISISPACE, ISIS On Board Computer. https://www.isispace.nl/product/on-board-computer/, (accessed 20.01.23).
[13] Xiphos Technologies, Q7 Processor. http://xiphos.com/products/q7-processor/, (accessed 20.01.23).
[14] Arm, Arm Firmware Framework for Arm A-profile. 30 November 2022, https://developer.arm.com/documentation/den0077/latest, (accessed 20.01.23).
[15] Trusted Computing Group, Homepage. https://trustedcomputinggroup.org/, (accessed 20.01.23).
[16] OASIS Open, OASIS PKCS 11 TC. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=pkcs11, (accessed 20.01.23).
[17] OASIS Open, Virtual I/O Device (VIRTIO) Version 1.2. 01 July 2022, https://docs.oasis-open.org/virtio/virtio/v1.2/virtio-v1.2.html, (accessed 20.01.23).
[18] GitHub, rv8 benchmark suite. https://github.com/michaeljclark/rv8-bench, (accessed 20.01.23).
[19] NASA, core Flight System (cFS). 16 June 2021, https://cfs.gsfc.nasa.gov/, (accessed 20.01.23).
[20] ESA, NanoSat MO Framework. https://nanosat-mo-framework.github.io/, (accessed 20.01.23).