Cybersecurity Approach for Multi-Tenant Small Satellites

Zain A. H. Hammadeh, Armin Purle-Kopacz, Hendrik Otte, Iasmina Dănescu, Andreas Lund, Daniel Lüdtke and Michael Felderer
Institute of Software Technology, German Aerospace Center (DLR), Germany

***Abstract*—As satellite systems become increasingly open to third-party software, ranging from student projects to commercial applications, the need for robust cybersecurity in space has become paramount. This paper presents a comprehensive security framework for executing untrusted code in space environments, focusing on software supply chain protection, virtualization for isolation, and multi-level Intrusion Detection and Response Systems (IDS/IRS).**

I. Introduction

Satellite-as-a-Service (SataaS) offers customers access to satellite capabilities without requiring them to own or operate the satellites themselves. It enables users to deploy and run their applications in orbit without the need for custom satellite hardware or dedicated missions. SataaS facilitates rapid, in-orbit testing, even for applications in early development stages. Both research-oriented and commercial applications can benefit from SataaS by significantly reducing costs and accelerating deployment timelines. A notable example is ESA's OPS-SAT 1 mission [1], which demonstrated the feasibility of allowing untrusted or low Technology Readiness Level (TRL) software to control a satellite. This was achieved by incorporating a secure, secondary on-board computer (OBC) that could take over control of the spacecraft in case the experimental software caused unsafe behavior. The ROSPIN-SAT-1 mission [2] aims to provide a streamlined, low-intervention sandbox platform for third-party experimenters to run untrusted and unsecured code in orbit. At the German Aerospace Center (DLR), we are developing the Stellar Apps project [3], which provides a secure and high-performance execution environment for on-board applications. Stellar Apps is designed to simplify access to space for low-TRL applications, while ensuring the safety and security of the host satellite.

However, enabling rapid prototyping, i.e., allowing unverified third-party software to run on operational satellites, introduces significant safety and security risks. Such applications may contain undetected bugs, security vulnerabilities, or even malicious code. Therefore, cybersecurity becomes a critical challenge for SataaS, in addition to other safety concerns inherent to satellite operations. SataaS inherits many architectural principles and goals from cloud computing, allowing us to adopt some well-established cybersecurity strategies. Nevertheless, the space environment imposes unique constraints: limited uplink bandwidth, restricted power and computational resources, and minimal thermal management options. These limitations necessitate the adaptation and redesign of cybersecurity solutions specifically for space systems. The importance of this topic has been emphasized in numerous studies (e.g., [4], [5], [6]), especially following the high-profile cyberattack on Viasat Inc.'s KA-SAT satellite network [7]. The research papers focused on exploring attack vectors on space systems and analyzing vulnerabilities. Willbold et al. proposed in [4] an analysis of satellite security and vulnerabilities for small satellites. The authors applied fuzzing techniques to detect vulnerabilities. Other researchers published their vision for secure space systems in the era of Commercially Off-the-Shelf (COTS) hardware and software, e.g.,[5]. Hammadeh et al. presented in [6] a security engineering perspective across the lifecycle of space systems. Recently, security testing has conducted on several major space-related software applications. These efforts has led to publish more than twenty CVEs (see Table I[1]) and security advisories in [8].

TABLE I
List of selected CVEs in Space Systems.

CVE	Product	Score
CVE-2024-44912	NASA Cryptolib	7.5 HIGH
CVE-2024-44911	NASA Cryptolib	7.5 HIGH
CVE-2024-44910	NASA Cryptolib	7.5 HIGH
CVE-2024-35061	NASA AIT-Core	7.3 HIGH
CVE-2024-35060	NASA	7.5 HIGH
CVE-2024-35059	NASA	7.5 HIGH
CVE-2024-35058	NASA	7.5 HIGH
CVE-2024-35057	NASA	7.5 HIGH
CVE-2024-35056	NASA	9.8 CRITICAL
CVE-2023-47311	YaMCS	6.1 MEDIUM
CVE-2023-46471	YaMCS	5.4 MEDIUM
CVE-2023-46470	YaMCS	5.4 MEDIUM
CVE-2023-45885	NASA Open MCT	5.4 MEDIUM
CVE-2023-45884	NASA Open MCT	6.5 MEDIUM
CVE-2023-45282	NASA Open MCT	7.5 HIGH
CVE-2023-45281	YaMCS	6.1 MEDIUM
CVE-2023-45280	YaMCS	5.4 MEDIUM
CVE-2023-45279	YaMCS	5.4 MEDIUM
CVE-2023-45278	NASA Open MCT	9.1 CRITICAL
CVE-2023-45277	YaMCS	7.5 HIGH

This paper addresses the challenges of securely hosting third-party applications on satellites. We present our approach

[1] You find them on https://nvd.nist.gov/vuln/detail/"CVE number", e.g., https://nvd.nist.gov/vuln/detail/CVE-2024-44912

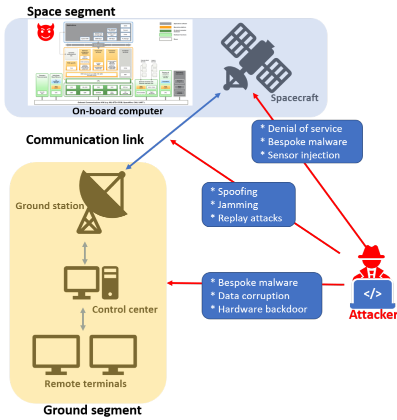

Fig. 1. Different space infrastructure segments may be subject to different attacks.

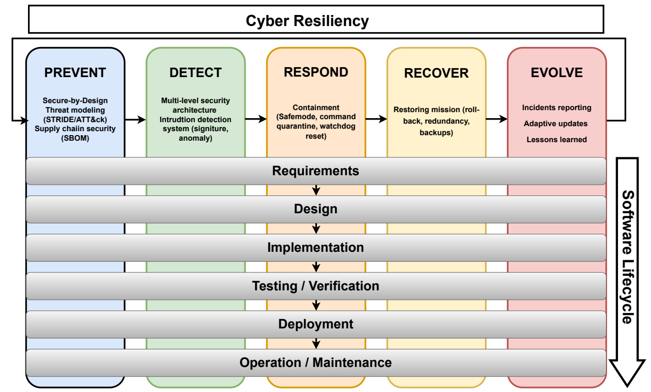

Fig. 2. Cyber resiliency across the software lifecycle.

in the context of the Stellar Apps platform, focusing on four key levels of security:

- Pre-deployment: Offline vulnerability analysis
- Application level: Execution isolation between applications
- Node level: Access control and application monitoring
- System level: Node health and behavior monitoring

The remainder of this paper is structured as follows: In the next section, SectionII, we present the need to act beyond prevention, i.e., the need for cyber resilient on-board software for spacecrafts. Section III presents the design of Stellar Apps as a Satellite-as-a-Service platform, providing all necessary background to keep the paper self-contained. Section IV outlines the threat model and potential security risks associated with hosting third-party applications. Our proposed security solutions are detailed in Sections V, VI and VII. Finally, Section VIII summarizes our findings and concludes the paper.

II. Cyber Resiliency

Building on the foundation established in the introduction, ensuring the security of space software requires more than preventive measures alone. Modern spacecraft rely on complex software ecosystems [9], integrating diverse tasks across multiple embedded systems with varying levels of criticality. These mixed-criticality environments often include both proprietary and third-party software components, including COTS modules, which can introduce vulnerabilities if not carefully managed [10], [11].

Cyber resiliency in space systems encompasses the capacity to anticipate, withstand, and recover from cyber incidents while maintaining mission-critical functions. As illustrated in Fig. 1, the primary targets include the ground segment, communication links, and the space segment itself. Each segment poses its own distinct security challenges: ground stations and mission control centers are critical hubs for satellite command and monitoring; communication links must resist spoofing, replay attacks, and jamming; while the spacecraft itself must operate reliably even when hosting third-party applications or executing on potentially vulnerable COTS hardware. Achieving cyber resiliency requires a lifecycle-driven approach, incorporating the principles of prevention, detection, response, recovery, and evolution. Preventive measures, implemented during system design and development, aim to reduce the attack surface through secure coding practices, threat modeling, and component hardening. Detection mechanisms, such as host-based, network-based, and hybrid intrusion detection systems (IDS), continuously monitor both software behavior and network traffic to identify anomalies in real-time [12], [13]. Once an incident is identified, response strategies—ranging from automated on-board interventions to ground-assisted decisions—seek to contain the threat while preserving operational continuity [14], [15]. Recovery processes then restore full functionality, leveraging redundancy, rollback mechanisms, and verified backups to resume nominal operations. Finally, evolution integrates lessons learned into the system design, updates detection rules, and adapts operational procedures, thereby increasing resilience over the mission's lifespan.

This comprehensive perspective on cyber resiliency can be represented as a continuous feedback loop across the software lifecycle, as shown in Fig. 2. In each lifecycle phase—requirements, design, implementation, testing, deployment, and operation—specific activities contribute to resilience. For instance, security and resiliency requirements identified during the concept phase are implemented through secure design patterns and runtime monitoring mechanisms. During operation, anomaly-based IDS models and telemetry monitoring support real-time threat detection, while automated recovery and redundancy measures ensure continuity of critical functions. Lessons learned from incidents and operational anomalies are subsequently fed back into design and maintenance, improving defenses and enabling adaptive system evolution.

III. General Design

The Stellar Apps software platform is developed in accordance with specific use cases and requirements. Its foun-

dational software is derived from the ScOSA project [16]. ScOSA (Scalable On-Board Computing for Space Avionics) is a distributed, heterogeneous on-board computing (OBC) architecture that integrates a radiation-hardened processor (specifically, a single GRE712RC LEON3FT in this project) with multiple commercial off-the-shelf (COTS) system-on-chip (SoC) units—namely, eight Xilinx Zynq 7000 devices—connected via a SpaceWire network.

Given the reliance on radiation-sensitive COTS components, ScOSA employs a middleware layer that enhances system reliability by detecting node failures and migrating tasks to other operational nodes, ensuring continued execution. If a failure is identified as transient, the affected node can be automatically reintegrated into the system.

Building upon ScOSA, the Stellar Apps project [3] extends these capabilities. The design of the development and execution environment in Stellar Apps reflects the lessons learned from ScOSA, addressing developer needs for a more flexible environment and broader support for development and runtime libraries. In particular, there is a strong demand for newer versions of AI frameworks such as PyTorch and TensorFlow. The platform also emphasizes rapid deployment to orbit and simplified update mechanisms to enable iterative development and in-orbit testing.

Stellar Apps inherits from the ScOSA the ability to run on RTEMS and Linux. We aim to enhance Linux with real-time extensions to support time-critical applications. A dedicated flight software core provides essential system services and supervises application execution, with the ability to terminate applications exhibiting non-nominal behavior.

Applications and their dependencies are packaged as containers. A central registry hosts a curated set of framework and library versions, removing the previous limitation—seen in the ScOSA flight experiment—that required all applications to use the same library versions. This container-based approach allows multiple application versions to coexist and execute concurrently. For AI workloads, both PyTorch and TensorFlow are supported. Beyond ScOSA's FPGA co-processor architecture, Stellar Apps also plans to integrate specialized AI FPGA designs [17] and embedded GPUs.

For interfacing with sensors, actuators, other on-board applications, and ground systems, the platform provides abstracted APIs that hide underlying communication protocols. This abstraction simplifies development by removing the need to implement specific telecommand and telemetry standards. The programming model aligns with that of lightweight web services.

Stellar Apps will be available on multiple platforms, including the GR712 and Xilinx Zynq 7000 (used in ScOSA and the CAPTn-1 mission), x86, and a new RISC-V-based reference platform.

Alongside the space segment environment, development and execution environments will be made available for application developers and for early-stage ground testing. A signature-based verification system ensures integrity and trust throughout the deployment process, enabling developers to test their applications in the same environment in which they will later operate in orbit.

Figure 6 illustrates the distributed architecture of Stellar Apps. Each node in the system runs the system management services responsible for communication, reconfiguration, and health monitoring. Additionally, an application manager on each node handles application-level communication and security monitoring. Stellar Apps is designed as a mixed-criticality execution platform, allowing high-criticality applications, such as attitude and orbit control systems (AOCS), to run alongside less critical applications without requiring dedicated hardware.

IV. Threat Modeling and Risk Assessment

Attacks can target various components of a space mission such as the ground segment, communication link, and space segment, and may involve supply chain compromise or unauthorized access to mission infrastructure [18]. Cyberattacks in particular affect both the data and the systems used to transmit and process it, and may involve malware, exploitation of legacy protocol vulnerabilities, or injection of false commands and data [19].

In some cases, the attack objective is clear—such as deploying ransomware [20]—but often, attribution remains difficult. While such attacks may not require extensive physical resources, they demand deep knowledge of the target systems. Their potential impact is significant, ranging from temporary service disruption to the total loss of control over a satellite or an entire constellation [19], [21].

This paper focuses specifically on cyberattacks introduced through the supply chain or by third-party applications. Supply chain threats primarily involve components such as open-source operating systems (OS), software libraries, and development tools (e.g., compilers). Adversaries may inject backdoors, ransomware, or other malicious code into these components, which can later be exploited to exfiltrate data or launch attacks—such as denial-of-service (DoS) operations. Similarly, third-party applications themselves may be compromised through their own supply chains, or may intentionally contain ransomware or malicious payload applications.

A prerequisite for Threat modeling and risk assessment (TMRA) is a clear definition of the system under analysis. Attack surfaces and attack vectors are not abstract concepts but are derived from the architecture itself. Attack surfaces correspond to the interfaces, resources, and components through which an adversarial entity can interact with the system, while attack vectors describe the means by which these surfaces may be exploited [22].

In recent years, the on-board software (OBSW) architectures have evolved from monolithic designs to layered and modular systems, as seen for example in the case of Nanosat MO Framework [23] used in the OPS-SAT mission. Functionality is distributed across multiple abstraction levels, typically including a general-purpose OS, middleware/platform service layer, and untrusted application-level components. While this decomposition improves flexibility and enables more advanced

functionality, it also introduces additional interfaces and dependencies that expand the system's attack surface. Fig.6 showcases this layered architecture approach in the case of Stellar Apps. In our use case, there are the following attack surfaces:

- *Platform firmware & configuration*: bootloader, secure boot chain, firmware image store, persistent configuration store.
- *Kernel & privilege enforcement*: OS kernel, privilege enforcement mechanisms, memory protection, resource control, device drivers.
- *Tenant runtime & isolation*: third-party application runtime, container runtime, container/sandbox configuration, inter-process communication (IPC) channels.
- *Platform management & services*: software update mechanism, execution scheduler, filesystem and storage, shared service logic, tenant-facing APIs, telemetry (TM) generation pipeline, telecommand (TC) processing pipeline.
- *Cyber-physical interfaces*: internal buses, payload hardware interfaces.
- *External interfaces*: command interface, file/experiment upload, authentication mechanisms, CI/CD and registry.

The next step is to move from *where*–attack surfaces–to *how*–attack vectors. The following attack vectors were selected for this mission class–based on their relevance to the described system–using the Common Attack Pattern Enumeration and Classification (CAPEC) taxonomy [24] and Space Attack Research and Tactic Analysis (SPARTA) matrix [25]:

- *Unauthorized code introduction*: software is deployed or executed on-board through upload, update, or supply-chain mechanisms without sufficient verification of its integrity, origin, or compliance with platform policies.
- *State persistence/recovery abuse*: malicious state is preserved across restart, update, or recovery cycles through configuration or persistent storage.
- *Privilege escalation*: a component operating within a restricted context gains access to higher privilege levels due to insufficient enforcement of access control or isolation policies.
- *Lateral movement*: a compromised component leverages shared communication mechanisms or insufficiently isolated resources to influence or access other components within the system.
- *Isolation boundary escape*: a violation of runtime isolation boundaries, allowing a component to access resources outside its intended execution domain; while this does not inherently grant full system privileges, it creates conditions under which privilege escalation and lateral movement may become possible.
- *Resource exhaustion*: shared computational or storage resources are consumed beyond intended limits due to insufficient enforcement of quotas, scheduling policies, or isolation mechanisms, resulting in degradation or disruption of system functionality.
- *Service abuse*: an untrusted application interacts with shared platform services through legitimate interfaces but violates intended usage policies due to insufficient authorization, validation, or resource control, enabling unintended influence over system behavior without breaching isolation boundaries; includes misuse of services under safe-mode conditions (during potentially reduced protection enforcement).
- *TM/TC path manipulation*: inputs to command handling, TM generation, or internal control logic are altered through exposed interfaces or shared communication mechanisms, resulting in unintended system behavior due to insufficient validation or trust assumptions.

Attack surfaces and attack vectors identified during threat modeling are not sufficient on their own for risk assessment, as they lack system-specific context. To ensure a consistent approach, scenarios were constructed using the following pattern: an attacker leverages a given attack surface to perform a specific attack vector, resulting in an undesired effect, which may cause a mission-level consequence. This structure provides the necessary context to evaluate likelihood and severity in a meaningful and system-specific manner. Two examples of constructed scenarios are described below.

Scenario I: Privilege Escalation

A deployed third-party application operates within a restricted execution environment with limited access to platform resources. By exploiting a vulnerability in the container runtime or a misconfigured IPC interface, the applications escapes its intended privilege boundary and gains access to higher-privilege execution context. From this position, it may interact with platform services or system resources that were previously inaccessible, such as TC processing pipeline, or the software management interface, potentially enabling further compromise of the platform or other co-hosted applications.

Assessment

In this scenario, privilege escalation requires a third-party application to identify and exploit a vulnerability or misconfiguration, such as those in the container runtime or IPC interfaces. While such vulnerabilities exist in practice, their exploitation is non-trivial and requires specific knowledge of the target platform. The container runtime and privilege enforcement mechanisms represent active architectural barriers, meaning the attack is feasible but not straightforward. Multiple conditions must align: the presence of an exploitable weakness, sufficient attacker knowledge, and the absence of detection, making this a medium **(score 3)** likelihood scenario.

A successful privilege escalation grants the attacker access to a higher-privilege execution context, from which more critical components may be reached. This represents a significant violation of privilege separation boundaries and creates conditions under which further compromise, including lateral movement or unauthorized command execution, becomes possible. However, privilege escalation alone does not constitute full platform compromise. It can represent a step toward that outcome rather than the outcome itself: meaningful architectural barriers remain between an escalated context and

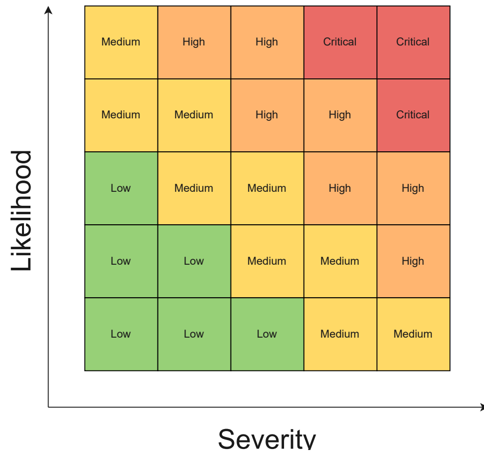

Fig. 3. Risk Matrix

complete control of the platform, and their exploitation would require additional steps beyond the scope of this scenario. Therefore, the impact threatens the integrity of the platform without necessarily reaching it, suggesting a major **(score 4)** rather than catastrophic severity rating. Therefore, the scenario is considered to be of **high risk** as Fig. 3 illustrates.

Scenario II: Service Abuse

A deployed third-party application interacts with shared platform services through legitimate, exposed interfaces. By issuing excessive or malformed requests (whether deliberately or as a result of a compromised or poorly implemented application) it overwhelms shared service resource. This does not constitute a violation of isolation boundaries, but it results in resource contention that may degrade the availability of platform services for other co-hosted applications, potentially disrupting mission-critical functions.

Assessment

Service abuse does not require exploitation of a vulnerability in the traditional sense: it relies on legitimate interfaces being used in an unintended manner. A third-party application has access to platform APIs by design, meaning the preconditions for this attack are inherent to the architecture. The primary barrier is rate limiting or request validation, which may be inconsistently enforced across platform services. The low enforcement barrier and the availability of legitimate access paths make this a high **(score 4)** likelihood scenario.

While service abuse can degrade platform performance and affect the availability of shared services, it does not violate isolation or privilege boundaries. The attack operates entirely within the allocated execution context of the application. Its impact is therefore confined to resource contention and service responsiveness. Critically, service abuse does not create conditions for further escalation or lateral movement, distinguishing it from scenarios where a boundary violation opens pathways to deeper compromise. Recovery is possible once the offending application is identified and terminated, with no permanent impact to architectural guarantees. The absence of any boundary violation and the lack of escalation potential limit the severity to minor **(score 2)**.Based on Fig. 3, this scenario is considered to be of **medium risk**.

To address such scenarios, our approach is based on three key mitigation strategies:

1) Pre-deployment verification – analyzing software for vulnerabilities and ensuring integrity before deployment.
2) Isolation and contracting – isolating application execution and defining strict contracts for behavior and resource usage.
3) Run-time monitoring – continuously supervising applications and system behavior to detect and respond to anomalies.

V. Pre-deployment Verification

Designing a platform capable of executing untrusted code in space requires careful consideration of the entire development and deployment pipeline. This pipeline includes three key components: (1) the Developer Segment, (2) the Ground Interface, and (3) the Space Segment (see Figure 4).

The Stellar Apps concept [3] foresees that applications are created by third-party developers, e.g. university teams, students, independent researchers, or commercial entities. However, these developers are not granted direct access to upload their applications to the spacecraft. Instead, a trusted intermediary, the Ground Interface, is introduced and acts as a safeguard between developers and the space segment.

This restriction is critical, as third-party developers may lack the expertise required for safe deployment to space and could unintentionally or intentionally upload malicious, unstable, or unverified code that endangers the mission. Therefore, all applications must first be submitted to the Ground Interface for a thorough verification process.

The Ground Interface performs various automated and manual checks, including:

- Static and dynamic code analysis
- Detection of known vulnerabilities
- Identification of hardware-specific bugs
- Behavioral profiling and anomaly detection

If issues are found, they can be addressed through collaboration with the developer or used to identify potentially malicious actors. Once the software passes these checks and both the Ground Interface and developer agree it is safe, the application is uploaded to the satellite via the designated ground station infrastructure, where it can be deployed and executed.

To ensure code authenticity and traceability, we propose a signature-based verification system using public-key cryptography. Each registered developer is issued a private signing key used to digitally sign their application packages. These signatures verify the origin of the code and allow for developer-specific entitlements—for example, only certain developers may be authorized to access sensitive components such as onboard cameras. This mechanism helps enforce the principle of least privilege, thereby minimizing the attack surface on

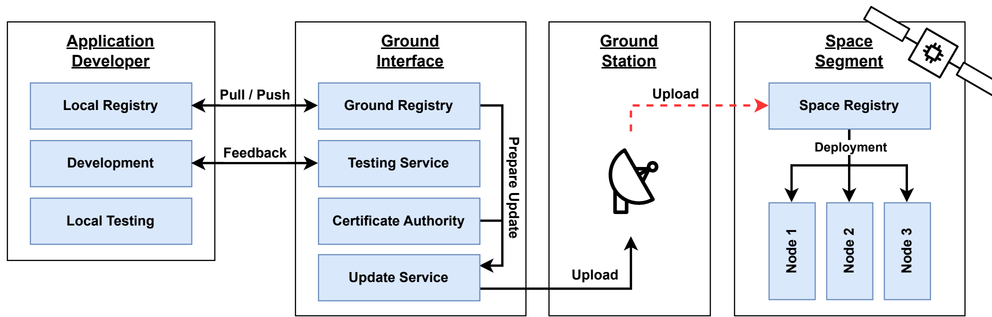

Fig. 4. Stellar Apps' schematic process of uploading/deploying an application to a satellite. [3]

the satellite. In addition, before deployment to space, the Ground Interface re-signs the validated application with a special execution key. Only code signed with this key is accepted for execution on the space segment. This added layer of authorization ensures that even if an attacker manages to bypass the Ground Interface and upload an application, it cannot be executed without a valid Ground Interface signature. For this system to function, the necessary public keys must be securely stored onboard the satellite.

It is important to note that even signed applications from trusted developers can contain malicious code, particularly due to supply chain attacks. These attacks exploit external tools or third-party libraries— including open source—that developers integrate into their applications. For instance, the recent attack on xz-utils [26] demonstrated how subtle backdoors can be embedded in widely used packages without the knowledge of the application developers. To mitigate this risk:

- External dependencies should be used only when necessary.
- All dependencies must be regularly tested and audited for potential compromise.

While it is not possible to eliminate supply chain risks entirely, a robust verification process, tight control over execution privileges, and continuous dependency monitoring form a strong defense-in-depth strategy for safely enabling untrusted software in space.

VI. Isolation and Contracting

Virtualization is the process of creating virtual environments that share the same underlying physical hardware. These environments can include virtual storage, servers, operating systems, and more. A key benefit of virtualization is isolation, which ensures that activities within one virtual environment remain invisible and unaffected by others. This isolation helps to prevent faults in one environment from propagating to others and enables the separation of critical resources such as CPU, memory, and disk space. Various technologies implement these principles, with three primary approaches highlighted here: hypervisors, sandboxing, and containerization.

One crucial aspect of our software platform is the deployment of applications in space, particularly when dealing with untrusted third-party code. Such applications need to be updated and replaced frequently. However, hosting multiple applications on the same system can introduce challenges, particularly when applications require conflicting dependencies, such as different versions of the same library. To address this, we adopt a containerization strategy. Containers allow applications to be packaged together with all their dependencies and runtime requirements, ensuring consistency and reliability across deployment environments. Applications can be deployed to the space system simply by transferring container images.

To ensure controlled execution, every application is governed by a formal contract managed by the Application Manager (Fig.5). This contract defines the application's operational boundaries, including resource allocation and inter-process communication privileges. Furthermore, it encapsulates the application's behavioral specifications, encompassing temporal constraints, compute utilization (CPU, memory, and GPU), and authorization for accessing hardware accelerators, sensors, and external services. Contracts are maintained in a location isolated from the application logic, utilizing machine-readable formats such as JSON, YAML, or XML, to facilitate automated processing.

While sandboxing and containerization provide strong OS-level isolation between applications and the host system, their security effectiveness depends heavily on the integrity of container images, proper configuration, and the security of the host kernel. Misconfigurations, such as running containers with elevated privileges (e.g., as root or with the SYS_ADMIN Linux capability), can break the isolation guarantees and expose the host to attacks [27]. Although containers operate in isolated user spaces, they share the host kernel, making kernel-level vulnerabilities a serious risk. Malicious applications could exploit such vulnerabilities to escape the container, gain root access, or compromise the entire system [27], [28]. For example, vulnerabilities in kernel memory management can bypass protections like Kernel Address Space Layout Randomization (KASLR) [29]. Similarly, attackers can abuse exception handlers to perform Denial-of-Service (DoS) attacks by exceeding resource limits, as demonstrated in [30]. The Linux kernel, which underpins most container engines, has

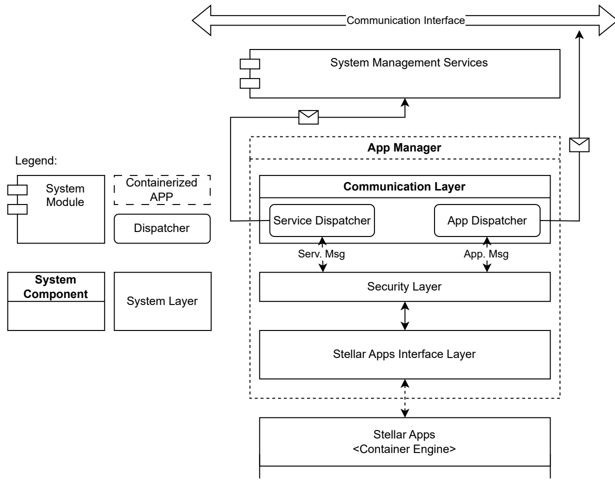

Fig. 5. The application manager represents the interface between the applications and the system management services. Also, it implements the node level intrusion detection and intrusion response mechanisms.

thousands of known vulnerabilities[2] and an unknown number of zero-day exploits. As such, attacks exploiting kernel flaws are not hypothetical, they are only a matter of time. To mitigate these risks, we add an access control mechanism and monitoring on the node and the system level.

In summary, while containerization offers many advantages for deploying untrusted code in space, its security relies on defense-in-depth: a combination of technical safeguards, strict policy enforcement, and continuous monitoring.

VII. Run-time Protection

Securing the communication link between the ground segment and the satellite is critical to defending against cyberattacks. Techniques such as end-to-end encryption can mitigate threats like spoofing and replay attacks. However, developers of on-board software must not assume that the satellite environment is inherently secure, especially in scenarios where satellites serve as execution platforms for third-party applications, which may be malicious or vulnerable by design. Virtualization technologies, such as sandboxing, containerization, and hypervisors, offer strong isolation guarantees between applications. Nonetheless, these mechanisms themselves can be targeted by cyberattacks [31], and their security depends heavily on proper configuration and the integrity of the host system. Another significant motivation for implementing cyber-resilient space systems is the threat of Denial-of-Service (DoS) attacks. These are typically simpler to execute than more sophisticated attacks but are often harder to predict. Of particular concern are sensor-disruption DoS attacks [32], which can severely impact the functioning of the software stack [33].

[2]https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendorid=33

To detect and mitigate such threats, efficient access control mechanisms and Intrusion Detection Systems (IDS) are essential. These systems monitor application traffic and behavior in real-time to identify malicious activities [12]. Additionally, a robust Intrusion Response System (IRS) is necessary to ensure continued operation of essential satellite services even under attack. This includes entering a fail-operational mode that isolates compromised components while keeping critical subsystems running. Given the limited computational resources aboard spacecraft, IDS and IRS solutions must be optimized for low-latency response, minimal resource consumption, and high reliability, while remaining resilient to evolving threats.

There are two primary methods used in designing IDSs:

- Knowledge-Based Intrusion Detection: Also known as signature-based or misuse-based detection. This method relies on predefined rules and patterns derived from known attacks. Observed events are compared against these signatures to detect intrusions.
  - Pros: High accuracy for known threats; low false positive rate.
  - Cons: Ineffective against unknown or zero-day attacks.
- Behavioral-Based Intrusion Detection: Also called anomaly-based detection [34]. This approach flags deviations from a predefined model of "normal" system behavior.
  - Pros: Capable of detecting novel or previously unknown attacks.
  - Cons: Prone to higher false positive rates, potentially flagging legitimate behavior as malicious.

Table II presents a comparison between the signature IDS and anomaly IDS.

Detecting an intrusion alone is insufficient, effective responses must be in place to counteract attacks and limit their impact. An Intrusion Response System (IRS) handles the mitigation of detected threats and manages the system's transition to a safe state.

The simplest response may involve switching the satellite to safe mode and transmitting telemetry to the ground station. However, autonomous responses are preferable in many scenarios, especially when communication delays or outages prevent timely ground intervention. To avoid overloading the system with numerous hard-coded responses, we advocate for generic, flexible mitigation strategies. One such strategy is reconfiguration-based response, commonly used in fault-tolerant space systems [11]. For instance, the system may reassign workloads, isolate faulty components, or reboot nodes to recover from compromise, as demonstrated in [35].

A. Our Approach: Multi-Level Security Architecture

Our system implements access control, intrusion detection (ID), and intrusion response (IR) at two levels:

1) Node-Level: At the node level, the Application Manager includes a security layer responsible for access control and knowledge-based intrusion detection (see Figure 5). Each

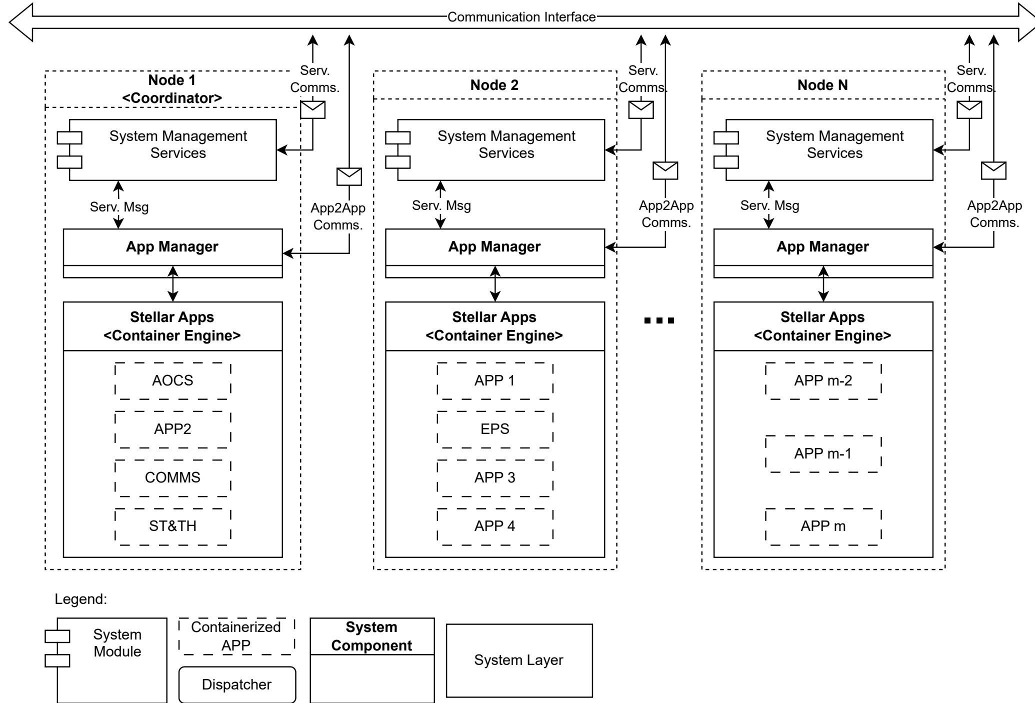

Fig. 6. Stellar Apps software architecture. Each node runs the system management services and the application manager besides the applications. From the Stellar Apps perspective, the basic on-board functionalities like AOCS are also applications.

TABLE II
A COMPARISON BETWEEN SIGNATURE IDS AND ANOMALY IDS FROM A SPACE SOFTWARE PERSPECTIVE.

Factor	Signature IDS	Anomaly IDS
Requirement type	Detect specific known attack	Detect unknown/new attacks
Application domain	OBC command validation, software updates, crypto checks	TC monitoring, payload data anomaly detection
Resource constraints [36]	Low CPU/memory, fits with embedded RTOS	Needs more memory/processing
False positives [36]	Very low	Higher (depends on training quality)
Adaptability	Static, rule-based	Dynamic, can evolve with new behaviors
Verification	Easy to trace to requirement	Harder to prove correctness (probabilistic)

message is checked for authorization: if an application attempts to access unauthorized resources or communicates with unauthorized destinations, it is flagged as suspicious. Crashing applications are also monitored. Upon detection, the IR component can terminate the offending application and generate telemetry containing logs for ground review. Listing 1 illustrates a pseudocode for the node-level ID-IR.

However, if an application is malicious enough to compromise the node, then more extensive measures are required.

2) System-Level: At the system level, a centralized ID-IR service runs on the coordinator node as part of system management services, see Figure 6. This service monitors the behavior of all nodes using behavioral-based intrusion detection. Metrics such as message frequency, message types, crash rates, and inter-node communication patterns are analyzed.

When malicious behavior is detected, the IR component can take several actions:

- Disable or power down the compromised node
- Trigger fail-operational or safe-mode configurations
- Notify the ground segment with detailed telemetry

Listing 2 illustrates a pseudocode for the system-level ID-IR. This layered architecture enhances the resilience of the entire platform by enabling both local and global perspectives on application behavior, thereby improving the accuracy of detection and enabling appropriate responses at different levels of severity.

```python
class NodeSecurityLayer:
    def __init__(self, access_policies):
        self.access_policies = access_policies
        self.suspicious_apps = set()

    def check_access(self, app_id, resource):
        if not self.is_authorized(app_id, resource):
            self.mark_suspicious(app_id)
            self.trigger_response(app_id)
        else:
            self.allow_access(app_id, resource)

    def is_authorized(self, app_id, resource):
        return resource in self.access_policies.get(
            app_id, [])

    def mark_suspicious(self, app_id):
        self.suspicious_apps.add(app_id)

    def trigger_response(self, app_id):
        kill_application(app_id)
        log_event("Unauthorized access by app:",
            app_id)
        send_telemetry("Security Alert", app_id)
```

Listing 1. Node-level IDS/IRS pseudocode

```python
class SystemMonitor:
    def __init__(self):
        self.node_metrics = {}
        self.anomaly_thresholds = {
            'crashes': 5,
            'msg_rate': 1000,
            'unauth_msgs': 10
        }

    def update_metrics(self, node_id, metric, value):
        if node_id not in self.node_metrics:
            self.node_metrics[node_id] = {}
        self.node_metrics[node_id][metric] = value
        self.analyze_behavior(node_id)

    def analyze_behavior(self, node_id):
        metrics = self.node_metrics[node_id]
        if (metrics.get('crashes', 0) > self.
            anomaly_thresholds['crashes'] or
            metrics.get('msg_rate', 0) > self.
                anomaly_thresholds['msg_rate'] or
            metrics.get('unauth_msgs', 0) > self.
                anomaly_thresholds['unauth_msgs']):
            self.respond_to_intrusion(node_id)

    def respond_to_intrusion(self, node_id):
        shutdown_node(node_id)
        log_event("Node shutdown due to anomaly:",
            node_id)
        enter_safe_mode()
        send_telemetry("System Alert", node_id)
```

Listing 2. System-level IDS/IRS pseudocode

VIII. Conclusion

The deployment of third-party applications in satellite systems introduces a new class of cybersecurity challenges. This paper has explored the vulnerabilities inherent in open space computing platforms and proposed a layered approach to mitigate these risks. By combining secure development pipelines, modular containerization strategies, and a two-level IDS/IRS architecture, we provide a resilient platform that enables untrusted code execution without compromising system integrity.

At the node level, access control and signature-based intrusion detection prevent unauthorized behaviors, while system-level anomaly detection monitors for signs of compromise across the entire constellation. We also highlight the importance of lightweight runtime monitoring and fault-tolerant response strategies like reconfiguration and safe-mode fallback.

Future work will involve implementing and testing this framework on representative hardware, evaluating its performance under real-time constraints, and exploring machine learning enhancements to the behavioral IDS.

References

[1] D. Evans and M. Merri, "OPS-SAT: A ESA nanosatellite for accelerating innovation in satellite control," in *SpaceOps 2014 Conference*, p. 1702, 2014.

[2] S. Severin, A. Purle-Kopacz, D. Bețco, I. Tincă, A. Stoica, N. Kelemen, I. Oprea, M.-A. Dobre, V. Ilincai, R. Tudorache, A. Dumitrescu, R.-C. Bișag, and M.-A. Niță, "Rospin-sat-1: Romania's first open source earth observation cubesat mission," in *Proceedings of the 2025 International Conference on Advanced Space Systems*, pp. 1–15, Curran Associates, Inc., 2025.

[3] H. Otte, A. Purle-Kopacz, K. Bleeke, A. Prat i Sala, Z. A. Haj Hammadeh, A. Lund, J.-G. Meß, M. Felderer, and D. Lüdtke, "Preliminary design of the stellar apps software platform for developing and executing on-board applications," in *Proceedings of the 2025 European Data Handling and Data Processing Conference for Space, EDHPC 2025*, Januar 2026.

[4] J. Willbold, M. Schloegel, M. Vögele, M. Gerhardt, T. Holz, and A. Abbasi, "Space odyssey: An experimental software security analysis of satellites," in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 1–19, 2023.

[5] N. Yadav, F. Vollmer, A.-R. Sadeghi, G. Smaragdakis, and A. Voulimeneas, "Orbital shield: Rethinking satellite security in the commercial off-the-shelf era," in *2024 Security for Space Systems (3S)*, pp. 1–11, 2024.

[6] Z. A. H. Hammadeh, M. Hamad, A. Olchawa, M. Starcik, R. Fradique, S. Langhammer, M. Hoffmann, F. Göhler, D. Lüdtke, M. Felderer, and S. Steinhorst, "Designing secure space systems," in *2025 Design, Automation & Test in Europe Conference (DATE)*, pp. 1–10, 2025.

[7] N. Boschetti, N. Gordon, and G. Falco, "Space cybersecurity lessons learned from the viasat cyberattack," in *AIAA Ascend*, pp. 1–8, 2022.

[8] VisionSpace, "List of publicly disclosed vulnerabilities for space systems." https://visionspace.com/category/cyber, 2024. Accessed: 2024-12-20.

[9] C. R. Prause, R. Gerlich, and R. Gerlich, "Fatal software failures in spaceflight," *Encyclopedia*, vol. 4, no. 2, pp. 936–965, 2024.

[10] D. Lüdtke, T. Firchau, C. G. Cortes, A. Lund, A. M. Nepal, M. M. Elbarrawy, Z. H. Hammadeh, J.-G. Meß, P. Kenny, F. Brömer, M. Mirzaagha, G. Saleip, H. Kirstein, C. Kirchhefer, and A. Gerndt, "ScOSA on the way to orbit: Reconfigurable high-performance computing for spacecraft," in *2023 IEEE Space Computing Conference (SCC)*, pp. 34–44, 2023.

[11] A. Lund, Z. A. H. Hammadeh, P. Kenny, V. Vishav, A. Kovalov, H. Watolla, A. Gerndt, and D. Lüdtke, "ScOSA system software: the reliable and scalable middleware for a heterogeneous and distributed on-board computer architecture," *CEAS Space Journal*, May 2021.

[12] M. Hamad, A. Finkenzeller, M. Kühr, A. Roberts, O. Maennel, V. Prevelakis, and S. Steinhorst, "REACT: Autonomous intrusion response system for intelligent vehicles," *Comput. Secur.*, vol. 145, Nov. 2024.

[13] M. Hamad, Z. A. H. Hammadeh, S. Saidi, V. Prevelakis, and R. Ernst, "Prediction of abnormal temporal behavior in real-time systems," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, (New York, NY, USA), p. 359–367, Association for Computing Machinery, 2018.

[14] Z. A. H. Hammadeh, M. Hasan, and M. Hamad, "RESCUE: A reconfigurable scheduling framework for securing multi-core real-time systems," *ACM Trans. Cyber-Phys. Syst.*, vol. 9, Aug. 2025.

[15] M. Hamad, Z. A. H. Hammadeh, D. Alessi, M. Hasan, M. Pese, D. Lüdtke, and S. Steinhorst, "Enhancing security through task migration in software-defined vehicles," *IEEE Internet of Things*, 2025. To Appear.

[16] D. Lüdtke, T. Firchau, C. E. Gonzalez Cortes, A. Lund, A. M. Nepal, M. M. H. H. Elbarrawy, Z. A. Haj Hammadeh, J.-G. Meß, P. Kenny, F. Brömer, M. Mirzaagha, G. Saleip, H. Kirstein, C. Kirchhefer, and A. Gerndt, "Scosa on the way to orbit: Reconfigurable high-performance computing for spacecraft," in *Proceedings - 2023 IEEE Space Computing Conference, SCC 2023*, pp. 34–44, August 2023.

[17] D. Helms, Q. Dariol, K. Grüttner, B. R. Perjikolaei, L. Einhaus, and S. Gregor, "Fpga based in-memory ai computing," in *ONERA Workshop on Advances in Artificial Intelligence for Aerospace Engineering*, pp. 1–2, Mai 2023.

[18] M. Manulis, C. P. Bridges, R. Harrison, V. Sekar, and A. Davis, "Cyber security in new space," *International Journal of Information Security*, vol. 30, pp. 287–311, 2021.

[19] K. Bingen, K. Johnson, M. Young, and J. Raymond, "Space threat assessment 2023," Apr. 2023. Online: https://www.csis.org/analysis/space-threat-assessment-2023.

[20] G. Falco, R. Thummala, and A. Kubadia, "WannaFly: An approach to satellite ransomware," in *2023 IEEE 9th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, pp. 84–93, 2023.

[21] Consultative Committee for Space Data Systems (CCSDS) , "Security threats against space missions," 2022. INFORMATIONAL REPORT. Online: https://public.ccsds.org/Pubs/350x1g3.pdf.

[22] National Institute of Standards and Technology, "Security and Privacy Controls for Information Systems and Organizations," Sept. 2020. U.S. Department of Commerce.

[23] C. Coelho, O. Koudelka, and M. Merri, "Nanosat mo framework: When obsw turns into apps," in *2017 IEEE Aerospace Conference*, pp. 1–8, 2017.

[24] The MITRE Corporation, "Common attack pattern enumeration and classification," 2023. Online: https://capec.mitre.org/index.html. Visited 29 April 2026.

[25] The Aerospace Corporation, "Space attack research & tactic analysis (sparta)," 2022. Online: https://sparta.aerospace.org. Visited 29 April 2026.

[26] "CVE-2024-3094.." CVE-ID CVE-2024-3094., 2024. Online: https://www.cve.org/CVERecord?id=CVE-2024-3094Visited on 29 April 2026.

[27] A. Y. Wong, E. G. Chekole, M. Ochoa, and J. Zhou, "On the security of containers: Threat modeling, attack analysis, and mitigation strategies," *Computers & Security*, vol. 128, p. 103140, 2023.

[28] Z. Jian and L. Chen, "A defense method against docker escape attack," in *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*, pp. 142–146, 2017.

[29] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, "A measurement study on linux container security: Attacks and countermeasures," in *Proceedings of the 34th annual computer security applications conference*, pp. 418–429, 2018.

[30] X. Gao, Z. Gu, Z. Li, H. Jamjoom, and C. Wang, "Houdini's escape: Breaking the resource rein of linux control groups," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1073–1086, 2019.

[31] G. Pék, L. Buttyán, and B. Bencsáth, "A survey of security issues in hardware virtualization," *ACM Comput. Surv.*, vol. 45, July 2013.

[32] Y. Tu, Z. Lin, I. Lee, and X. Hei, "Injected and delivered: Fabricating implicit control over actuation systems by spoofing inertial sensors," in *27th USENIX Security Symposium (USENIX Security 18)*, (Baltimore, MD), pp. 1545–1562, USENIX Association, Aug. 2018.

[33] A. Roberts, M. Malayjerdi, M. Bellone, R. Sell, O. Maennel, M. Hamad, and S. Steinhorst, " Analysis of Autonomous Driving Software to Low-Level Sensor Cyber Attacks ," in *2025 IEEE/ACM 20th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, (Ottawa, Canada), pp. 122–132, IEEE Computer Society, April 2025. to appear.

[34] M. Hamad, Z. A. H. Hammadeh, S. Saidi, V. Prevelakis, and R. Ernst, "Prediction of abnormal temporal behavior in real-time systems," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, (New York, NY, USA), p. 359–367, Association for Computing Machinery, 2018.

[35] Z. A. H. Hammadeh, M. Hasan, and M. Hamad, "Rescue: A reconfigurable scheduling framework for securing multi-core real-time systems," *ACM Trans. Cyber-Phys. Syst.*, Apr. 2025. Just Accepted.

[36] V. K. Ravindran, S. S. Ojha, and A. Kamboj, "A comparative analysis of signature-based and anomaly-based intrusion detection systems," *International Journal of Latest Technology in Engineering Management & Applied Science*, vol. 14, p. 209–214, Jun. 2025.