MORA

Hunting Space Bugs in your Sleep

Vissarion Moutafis

Delft University of Technology

TU Delft

MORA

Hunting Space Bugs in your Sleep

by

Vissarion Moutafis

Vissarion
Moutafis

Instructor:	G. Smaragdakis
Daily Supervisor:	A. Voulimeneas
ESA Supervisors:	A. Atlasis, N. Finne
Project Duration:	10, 2024 - 04, 2025
Faculty:	Faculty of Electrical Engineering, Mathematics and Computer Science , Delft

Cover:	Generated by ChatGPT
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

Preface

This work, for it is but a harmless thesis, was completed during one of the most peculiar and challenging periods of my life. It became my constant companion through countless sleepless nights, a relentless yet rewarding pursuit.

First of all, I would like to express my deepest gratitude to my supervisors; George, for his inspiration and unwavering motivation to push me toward excellence; Alex, for being not only a brilliant mentor but also a great friend; and Antonis, for his invaluable guidance and the trust he placed on me throughout this work.

I also thank the advisors on the European Space Agency for guiding this effort and also for facilitating and assisting our experimental work. I am very grateful. In addition, I would like to thank the people from TEC-SEC and TEC-EDD, especially Nick Panagiotopoulos, for allowing us to use the Cube-FlatSAT and have fun running our experiments on a realistic mission platform!

Above all, I want to thank my family and friends for their unwavering support. Their encouragement gave me the strength to persevere and see this work through to the end.

Vissarion Moutafis
Delft, April 2025

I love sleep; is my favorite...

Kanye West

Abstract

This thesis explores the security of On-Board Software (OBSW) within mixed-criticality space applications, emphasizing post-exploitation threats and the need for a structured and standardized vulnerability discovery and assessment framework.

To address these challenges, we develop a threat model tailored to RTOS-based space systems, identifying key attack surfaces and adversary capabilities. Our methodology leverages fuzzing methods to systematically uncover vulnerabilities in FreeRTOS, a widely adopted RTOS in space applications, and automates the false-positive/duplicates elimination procedure to minimize the manual work needed during crash triage. The results highlight weaknesses in task isolation and privilege management, demonstrating the feasibility of horizontal lateral movement within on-board software systems.

To evaluate the severity of identified vulnerabilities we integrate an adaptation of the Common Vulnerability Scoring System (CVSS) tailored to space software security with focus on the temporal and environmental metrics. Additionally, we validate our findings through a Cube-FlatSAT experimental setup, demonstrating real-world applicability and reinforcing the need for improved isolation mechanisms in space-grade RTOS.

This research also contributes to the SPACE-SHIELD framework by refining post-exploitation analysis techniques. Our work underscores the necessity of standardized security assessments for on-board space systems, making the first step for robust development of space software against emerging cyber threats in the space domain.

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
ESA	European Space Agency
CIA	Confidentiality, Integrity, Availability
LEO	Low Earth Orbit
GEO	Geostationary Orbit
GS	Ground Segment
SS	Space Segment
US	User Segment
TC	Telecommand
TM	Telemetry
CDHS	Command and Data Handling System
PHDS	Payload Data Handling System
COM	Communications Module
PLCOM	Payload Communication Module
ADCS	Attitude Determination and Control System
EPS	Electrical Power System
COTS	Common-off-the-shelf
MCS	Mixed Criticality System
MCA	Mixed Criticality Application
OBC	On Board Computer
OBSW	On Board Software
FDIR	Fault Detection, Isolation and Recovery
OS	Operating System
IPC	Inter-Process Communication
RTOS	Real-Time Operating System
PoC	Proof Of Concept
ECSS	European Cooperation for Space Standardization
CCSDS	Consultative Committee for Space Data Systems
CVSS	Common Vulnerability Scoring System
CVE	Common Vulnerabilities and Exposures
IPC	Inter-Process Communication
MPU	Memory Protection Unit

1 Introduction

In recent decades, there has been a significant increase in the number of satellites orbiting the Earth, driven by advancements in telecommunication networks, navigation, earth observation [53]. The New Space era encourages the use of Commercial Off-The-Shelf (COTS) components, for both Low-Earth-Orbit (LEO) and deep space missions [6, 54]. This approach improves availability and reduces development costs, making it attractive for high-volume production, such as satellites used in 5G and upcoming 6G networks. In this work, we are particularly interested in the software technologies running on the space segment, namely the On-Board Software (OBSW) which runs on the On-Board Computers of different satellite compartments such as the Command and Data Handling System (CDHS) and Attitude Determination and Control System (ADCS).

The usage of Mixed Criticality systems is very appealing in the case of On-Board Computer (OBC) and Software (OBSW) [54] development and is needed for performing crucial operations for the satellite mission. Using hard real-time systems and hypervisors, satellite systems ensure the strict scheduling needed by mission operations supported by multi-processor hardware that implements the different levels of criticality into a single OBC component. Nevertheless, the security behind such software and hardware, which was heavily relied on the fact that OBSW was not reachable by design and because of the security enforced in the communication protocols, is now accessible more than never due to a combination of third-party hosted dependencies and COTS used in extensively in space deployments. Though developers are making adjustments on COTS to comply to space standards, this situation has opened new possibilities for attacks on space systems such as supply chain attacks and malicious payload propagation from the User Segment [14, 53, 54]. Although there is some preliminary research work on attacking space systems, more work needs to be done on the general security of OBC and OBSW components with respect to Confidentiality, Integrity and Availability triad (CIA) and isolation capabilities of space OS.

Real-Time Operating Systems (RTOS) are foundational components of the OBSW stack, playing a pivotal role in both satellite bus and payload subsystems. Their ability to provide hard real-time scheduling ensures that time-critical mission operations are executed reliably and predictably. Widely used RTOS, such as FreeRTOS [15], RTEMS [34], LynxOS [25], and VxWorks [51], support diverse applications across LEO and GEO satellites. For instance, missions like ESTCube-1 [39], Galilleo and Fermi Gamma-Ray Space Telescope have leveraged RTOS to facilitate efficient task scheduling and resource management. These systems must ensure robust isolation of high-priority tasks, preventing interference or data leakage between processes. Furthermore, they must guard against scenarios where an onboard attacker, having compromised a single task, escalates privileges or disrupts the entire system. Strengthening the resilience of RTOS against such threats is essential to maintaining the integrity and safety of modern space missions.

In this work, the main focus is on analyzing the security challenges faced by space OS for mixed-criticality applications. Though the work on OBSW security addresses some specific use cases [53], there is a need for standardized space OS security testing which ensures the systems robustness with

respect to inter-task communication, task scheduling and RTOS/hypervisor isolation issues. Attackers that successfully exploit such issues might compromise the CIA of different components and could possibly seize the control of a satellite system, causing significant damage to the targeted organization.

This research focuses on onboard attackers and aims to address security challenges in RTOS systems used in mixed-criticality space applications. The primary objectives are to identify and assess vulnerabilities and propose mitigations that align with established space security and safety standards, such as ECSS.

More specifically, we pose the following question, **RQ: "How can we standardize and automate vulnerability discovery and assessment in space OS for mixed criticality applications running on a satellite's OBC?"**. To better bind our research, we also pose the following sub-questions:

- **RQ1: "How can a robust threat model be defined for on-board software security in mixed-criticality space applications?"**. This involves identifying assets, actors and assumptions about the attacker and platform.
- **RQ2: "What security vulnerabilities exist in Real-Time Operating Systems used in space missions?"** This question focuses on discovering and assessing the security weaknesses within these critical software components.
- **RQ3: "How can we effectively use dynamic analysis to automate the assessment?"** The question shift the focus to the effectiveness of the implementation of a general framework that also handles scalable automation and elimination of false positives.
- **RQ4: "How applicable are our results in a representative set-up?"** This question is vital to understanding the actual impact of any findings discovered in a simulated environment against a real platform.

The research methodology begins with the development of a comprehensive threat model and risk assessment framework. This framework forms the foundation for identifying critical attack vectors and guiding the testing phase. To uncover vulnerabilities, we employ widely used techniques such as fuzzing. The discovery process is conducted in a simulated environment, which simplifies the creation and refinement of proof-of-concept (PoC) exploits. Once developed, these PoCs are tested on representative hardware to evaluate their reproducibility in environments that closely resemble real-world deployments, ensuring practical relevance and applicability.

The main contributions of this work lie in advancing the security of SpaceOS by addressing key challenges in Real-Time Operating Systems (RTOS) technologies, particularly in the context of mixed-criticality space systems. This research defines a comprehensive threat model tailored for the SpaceOS domain, capturing the unique needs of mixed-criticality environments and identifying attack surfaces specific to RTOS. One focus is on refining lateral movement attack paths, detailing how an attacker with an initial foothold could compromise additional system components, jeopardizing the satellite's functionality and security.

Security testing is conducted on SpaceOS, leveraging custom tools and methodologies to uncover vulnerabilities. Findings are eventually mapped to techniques from the current versions of SPACE-SHIELD [12] and SPARTA [43] frameworks and a new refined version of the former, ensuring alignment with established security standards and facilitating consistent vulnerability management. To evaluate the criticality of discovered vulnerabilities, we employ the CVSS [9] standard, quantifying their potential impact on satellite operations. For a more comprehensive application of CVSS scoring, we also redefine the different metrics so that they relate with the MSC/SpaceOS domains. Furthermore, the research discusses mitigations for various attack scenarios, trying to initiate the conversation for integrating terrestrial defense systems in the resource-constrained field of space systems.

This thesis is organized as follows, after this chapter, Chapter 2 provides the necessary background to contextualize the study, covering mixed-criticality applications, satellite systems, and relevant security concepts. Chapter 3 defines the threat model and the risk assessment framework, forming the foundation for subsequent analysis. Chapter 4 outlines the methodology for testing each identified attack vector, providing high-level guidance. Chapter 5 presents the experiments, evaluates vulnerabilities, and maps the findings to the SPACE-SHIELD framework [12]. Finally, Chapter 6 discusses contributions to SPACE-SHIELD, addresses broader SpaceOS design issues, proposes mitigation strategies, and highlights directions for future work.

2

Background

2.1. Space Standards and Existing Security Frameworks

European Cooperation for Space Standardization. The European Cooperation for Space Standardization is a collaboration between ESA, the European Space Industry, and several space agencies. It refers to companies that carry out space activities and establishes a coherent standard to govern the management, engineering, product assurance, and sustainability of space projects [44]. It is adopted by ESA and all the collaborators looking to procure a space project.

SPACE-SHIELD and SPARTA SPACE-SHIELD is a ATT&CK like framework providing users with security threat knowledge for Space Systems [12]. It focuses on covering a broad range of attack techniques and is inspired by the MITRE ATT&CK framework. It is created by ESA and is aimed at cybersecurity experts that want to create threat models and perform a structured and guided test of a space system. SPARTA is a similar framework addressing attack methodologies in the aerospace sector aiming to standardize threat modeling and security testing of spacecrafts [43]. It is created by The Aerospace Corporation and is part of their space security solutions.

Common Vulnerability Scoring System. The Common Vulnerability Scoring System [9] is a vulnerability assessment framework which is used to evaluate the severity of a vulnerability. Three distinct metric groups, namely Base, Temporal and Environmental, are used to conclude on a severity score. Base metrics have a 0-10 scale and are applicable to all domains, while Temporal and Environmental scores are domain specific and can modify the Base metric resulting into a final severity score and a CVSS vector, which is a textual representation of the values used to derive the score. Base Metrics consist of exploitability, privilege required, scope and CIA impact scorings, which are necessary to evaluate any vulnerability in any domain. Temporal Metrics are concerned with the maturity of the exploit technique, the mitigation status and the confidence of the applicability of such a vulnerability in different use cases with respect to the target domain. Finally, Environmental scores are dependent on the importance of the affected asset to the organization. They complement the importance of CIA features, modify the base metrics for enhancing the validity of the severity score for the target domain and organization while also considering existing mitigations for the discovered issues.

Though the existing standards and guides for security testing and threat modeling of space systems are developing and improving gradually, they cover a broad range of attacking techniques and remain quite high level with respect to mission specific attacks and spaceOS focused attacks. Comparing them to the MITRE ATT&CK [28] and MITRE EMB3D, it would be beneficial to improve threat categorization and even add a mapping to the components tested. Finally, CVSS scoring mechanism, though it is perfect for our purpose, needs additional refinements so that space developers can apply it on security findings in an educated manner, considering the MCS/SpaceOS domain.

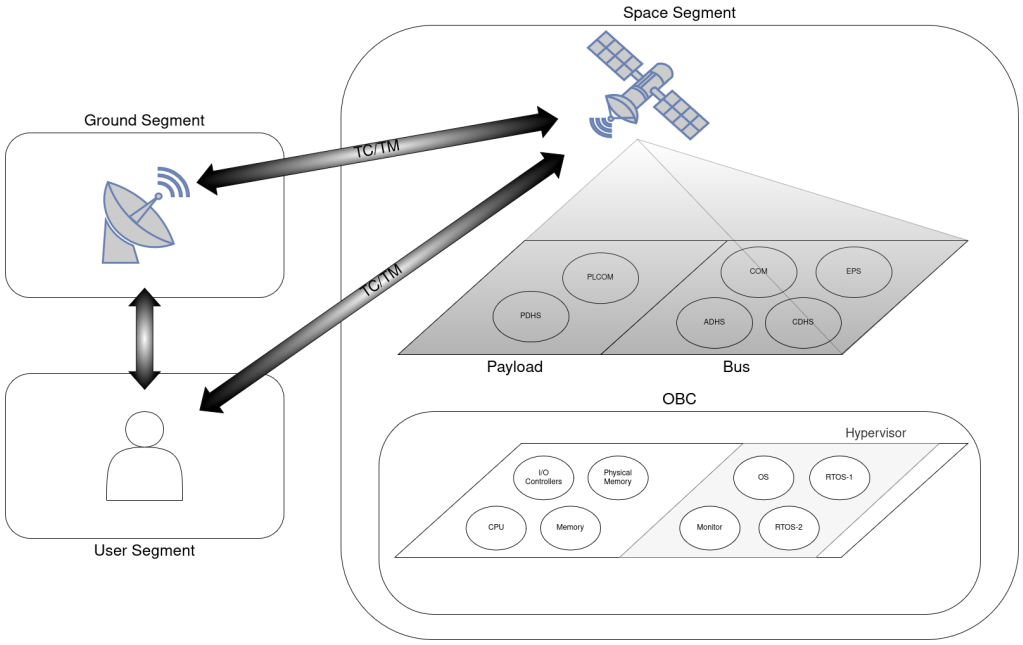

Figure 2.1: The different segments that collaborate in a mission. Space Segment consists of two main parts, the Bus and Payload. The Bus contains the Command and Data Handling System (CDHS), Communication Module (COM), Attitude Determination and Control System (ADCS) and Electrical Power System (EPS), while the satellite's Payloads contains mission specific components such as Payload's Communication Module (PLCOM) and Payload Data Handling System (PHDS). The computing components, e.g. CDHS contain an OBCs that contain hardware components and software (OBSW) which consists of the hypervisor and applications or general purpose OS runing on top of the former.

2.2. Satellite Systems and Software

In this section, we first provide an overview of the segment level modules, then we focus on the in-scope components of the space segment and their submodules. The design of the systems, their utility and the inter-component interaction is extensively discussed in the state-of-the-art literature regarding space security [14, 33, 53, 54, 55] and it is refined in great detail in the SAVOIR [35] and ECSS [44] standards.

Satellite systems are complex infrastructures, categorized into distinct segments to ensure effective operation and mission success: the Ground Segment, Space Segment, and User Segment. These components work in unison, fulfilling mission requirements across various orbits, such as Low Earth Orbit (LEO), Medium Earth Orbit (MEO) and Geostationary Orbit (GEO).

Low Earth Orbit (LEO), situated between 250 to 2,000 km above Earth, is predominantly utilized for Earth observation and CubeSat deployments due to its proximity and reduced latency. Conversely, Geostationary Orbit (GEO), at 35,786 km, is ideal for telecommunications as satellites maintain a fixed position relative to Earths surface, enabling uninterrupted coverage over specific areas.

The **Ground Segment** (GS) serves as the operational hub, encompassing ground stations and control centers. Operators manage satellite functions through Telecommand (TC) and receive Telemetry (TM) data regarding the satellites status and health. Ground stations utilize established communication protocols like those defined by the CCSDS to ensure robust data exchange.

The **Space Segment** (SS) consists of all space-based assets, including the satellite platform and payload. The platform supports essential functions such as power supply, attitude control, and communication, while the payload carries mission-specific instruments like cameras for Earth observation or radio systems for telecommunications.

The **User Segment** (US) represents the end-users who benefit from satellite services, ranging from GPS receivers to Earth observation data consumers. These segments may overlap in advanced systems, such as platforms operating with multi-tenant architectures, where satellite resources are shared among users [14, 53].

Key subsystems within the space segment include the Command and Data Handling System (CDHS),

which orchestrates all satellite operations via onboard software; the Communications Module (COM) for TC/TM traffic management; the Attitude Determination and Control System (ADCS) for orientation adjustments; the Electrical Power System (EPS) to manage power; and the payload, housing mission-specific tools and data handling systems.

Figure 2.1 showcase the segments and components which form the backbone of satellite missions, enabling seamless operation across Earth and space. Additionally, we also provide a detailed illustration of the Space Segment, consisted of the Bus and Payload, and distinguish on the different crucial components, especially in the OBC.

2.2.1. Payload Data Handling System

The Payload Data Handling System (PDHS) is a subsystem dedicated to managing and processing data generated by the satellite payload. As the primary interface between the payload and other satellite components, the PDHS ensures that mission-critical data is collected, processed, and transmitted efficiently. It is particularly significant in missions requiring high data throughput, such as Earth observation, satellite communication systems or scientific experiments [54].

During a mission, the PDHS collects raw data from the payload, applies necessary processing or compression algorithms, and prepares the data for transmission to the Ground Segment. It also supports data storage and retrieval, ensuring mission continuity in scenarios where real-time transmission is not feasible. For example, in deep-space earth observation missions, the PDHS stores data for later downlink when the satellite is in range of a ground station [14, 53].

The PDHS consists of several high-level software components:

- **Data Acquisition Software**: Interfaces with the payload to collect raw data.
- **Data Processing Software**: Performs compression, encryption, or formatting as required by the mission.
- **Storage Management Software**: Manages onboard data storage and retrieval systems.
- **Transmission Control Software**: Prepares data for downlink via communication subsystems.

2.2.2. Command and Data Handling System

The Command and Data Handling System (CDHS) is a critical subsystem in satellite architecture responsible for the coordination and execution of mission tasks. It acts as the brain of the satellite, managing the flow of data between various subsystems and the proper operation of the satellite throughout different phases of the mission. The CDHS facilitates communication between the satellite's payload and its supporting bus/platform, ensuring seamless operation during the mission [2, 35].

On the hardware level, as showcased in Figure 2.2, the On-Board data handling platform consists of various devices which define the computational domain of the CDHS [2], namely

- On-Board computers (OBC), which define the core computational power of the CDHS
- Remote Terminal Computers, performing discrete and analog data acquisition
- Platform Solid State Mass Memories, usually used in Earth observation, facilitating storage of large data objects
- TC/TM Units, handling the Telemetry traffic and propagating commands to the OBC when necessary
- Internal Command and Control Busses, such as the CAN Bus used for on-board communication between CDHS components

As displayed in Figure 2.2, the CDHS software layer typically comprises several functionalities [2, 35]:

- **System Management Software**: Implementation of the basic functionalities of the core space OS, handling memory management, I/O operations, etc.
- **Sensor Drivers**: Implementation of the interface layer with the on-board sensor hardware.
- **Data Processing Software**: Manages telemetry and payload data, ensuring proper formatting and transmission.

- **Fault Detection, Isolation, and Recovery (FDIR)**: Identifies anomalies, isolates faults, and triggers corrective actions.
- **Communication Management Software**: Handles telecommand (TC) and telemetry (TM) operations, adhering to CCSDS protocols.

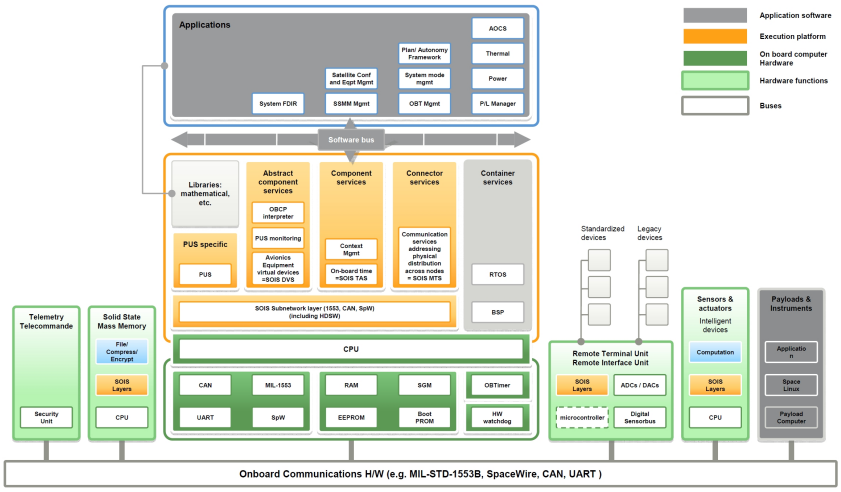

Figure 2.2: Detailed analysis of the On-Board Data Handling System components from SAVOIR [2, 35]. We can see the vast complexity of the hardware necessary to facilitate all of the CDHS mission requirements for keeping the satellite running and operating over mission data.

2.2.3. On-Board Computer and Software

The Onboard Computer (OBC) and Onboard Software (OBSW) form the central processing unit and operational software stack of a satellite. The OBC provides the computational power required to execute mission tasks, while the OBSW consists of RTOS, Board Support Packages (BSPs) and other components that orchestrate the resoure management, fault detection and management and execution of mission critical operations. Together, they act as the operational backbone of the satellite, ensuring compliance with space system engineering principles such as modularity, determinism, and failure recovery as defined by ECSS-E-ST-40C (Software Engineering) and SAVOIR guidelines [10, 35].

The OBC and OBSW reside within the Space Segment, interfacing directly with satellite subsystems via the onboard bus and indirectly with the Ground Segment through the Communication Module (COM) using telecommands (TC) and telemetry (TM) links. The communication flow enforces integrity verification and authentication mechanisms to prevent unauthorized command injection or spoofing attacks [40, 41]. Despite these protections, vulnerabilities in onboard software could still be exploited via privilege escalation, supply chain attacks or malicious updates, insufficient access control mechanisms or memory corruption flaws [53, 54].

The integration of Commercial Off-The-Shelf (COTS) components in OBC hardware and OBSW libraries is driven by cost efficiency, reduced development time, and the need for industry-wide standardization. However, ECSS-Q-ST-80C (Software Product Assurance) mandates rigorous validation and adaptation of COTS software to ensure compliance with mission-critical security and safety requirements [11]. The reliance on third-party software increases the attack surface, especially if the software lacks memory protection, is susceptible to privilege escalation, or fails to meet deterministic execution constraints required in real-time space applications. SAVOIR promotes a standardized avionics architecture, reducing security risks from heterogeneous COTS components by enforcing strict interface definitions and modular partitioning of software functions [35].

Given their pivotal role in mission operations, the OBC and OBSW can constitute as prime targets for cyber-attacks. A compromise in these systems could result in privilege escalation, unauthorized command execution, or software-based denial-of-service (DoS) attacks. While the ECSS risk assessment framework incorporates Fault Detection, Isolation, and Recovery (FDIR) mechanisms to mitigate some failures, a sufficiently sophisticated attack could still disrupt key satellite functions. For instance,

tampering with OBSW task scheduling could delay or disable payload operations, while a targeted attack on the OBC could interfere with power distribution, navigation, or maneuver execution, potentially jeopardizing satellite stability [14, 54].

A well-designed security framework must consider space system resilience by integrating hardware-enforced privilege separation (e.g., MPU-based task isolation), runtime integrity verification, and real-time intrusion detection systems. Without these safeguards, adversaries could exploit OBC and OBSW vulnerabilities to gain unauthorized access, manipulate onboard processes, or degrade system availability. Therefore, a standardized security testing approach is necessary to evaluate onboard software against both terrestrial and space-specific attack vectors, ensuring compliance with ECSS and SAVOIR standards [10, 11, 35].

2.3. Space Operating Systems

In this section, we discuss the significance of Real Time OS in the OBSW and the criteria which are used to determine the suitability of the underlying OS during the software design phase.

2.3.1. Real-Time Operating Systems

Real-Time Operating Systems (RTOS) are specialized operating systems designed to manage hardware resources and execute tasks with precise timing constraints. Unlike general-purpose operating systems, RTOS prioritize deterministic behavior, ensuring that critical tasks are executed within strict deadlines. They achieve this by implementing scheduling algorithms, such as rate-monotonic or earliest-deadline-first, which are tailored for real-time applications. This deterministic execution makes RTOS essential for systems where timing, reliability, and predictability are paramount [53].

In space systems, RTOS are integral to both the satellite bus and payload, enabling real-time processing of mission-critical tasks. They are widely used across various subsystems, such as the Command and Data Handling System (CDHS), the Attitude Determination and Control System (ADCS), and the Payload Data Handling System (PDHS). The unique requirements of space missions such as high reliability, resource constraints, and the need for fault tolerance make RTOS the preferred choice for managing satellite operations [14, 53].

RTOS are implemented in both custom and commercial off-the-shelf (COTS) configurations. Commonly used RTOS in space applications include FreeRTOS [15], RTEMS [34], LynxOS [25], and VxWorks [51]. These systems are tailored to the specific needs of space missions:

- **Task Scheduling**: RTOS prioritize high-criticality tasks, ensuring timely execution while managing lower-priority operations.
- **Memory Management**: RTOS allocate and protect memory to prevent interference between processes, which is crucial in mixed-criticality applications.
- **Inter-Process Communication (IPC)**: RTOS facilitate secure communication between subsystems, ensuring data integrity and synchronization.

The adoption of COTS RTOS solutions reduces development costs and time [54] but increases the potential of introducing vulnerabilities met in terrestrial systems. Despite these challenges, RTOS remain indispensable for enabling the precision, reliability, and resilience demanded by modern space missions.

2.3.2. Suitability for Space Missions

Although there are different space OS that could be used in space missions, the developer must consider various aspects of the RTOS in use to make sure that the software used meets the needs of a specific mission. The aforementioned systems offer distinct advantages and trade-offs, making them suitable for different mission requirements.

Flexibility FreeRTOS and RTEMS both being open-source and highly customizable, allow developers to tailor them to specific applications. FreeRTOS provides minimal RTOS functionality, making it lightweight and easier to adapt, while RTEMS offers a broader feature set, closer to proprietary solutions like VxWorks. VxWorks, though closed-source, provides extensive customization options for

secure multi-threaded applications, whereas LynxOS, with its deterministic and non-modular design, emphasizes reliability over flexibility.

Performance FreeRTOS and RTEMS are both known for low overhead and fast execution [15, 34], comparable to VxWorks in certain use cases. LynxOS is optimized for predictability, ensuring consistent response times even under heavy I/O, leveraging its kernel's threading model. FreeRTOS, with its lightweight design, is particularly advantageous for resource-constrained systems, while RTEMS provides similar performance metrics for more feature-intensive applications.

Security considerations vary significantly across these RTOS. FreeRTOS and RTEMS lack inherent privilege separation unless paired with hardware features like an MPU or MMU [27], which can be a limitation for high-assurance systems. However, their open-source nature fosters extensive testing and collaborative security improvements. In contrast, LynxOS and VxWorks, both microkernel systems, provide robust security with kernel-user mode separation and modular middleware. LynxOS further isolates running applications into VM-like partitions for memory, time, and resource separation, making it highly resilient to external threats.

When it comes to CPU Architecture **Support**, FreeRTOS leads with support for a wide array of architectures, including ARM, RISC-V, x86, and many more, making it highly versatile [15]. RTEMS supports an impressive range, including LEON and PowerPC [34], often utilized in aerospace applications. LynxOS and VxWorks offer robust architecture support, focusing on commonly used platforms such as ARM, x86, and PowerPC, with LynxOS excelling in environments requiring deterministic behavior [25, 51].

FreeRTOS and RTEMS are excellent choices for projects prioritizing open-source flexibility and low resource requirements, while VxWorks and LynxOS are more suitable for secure and deterministic applications where modularity and resilience are critical, given that the version used is compliant with space standards.

2.4. Fuzzing

Fuzzing is a dynamic software testing technique that involves providing a program with a wide range of random, malformed, or unexpected inputs to identify vulnerabilities, crashes, or unexpected behavior. Unlike static analysis, which inspects the code without execution, fuzzing executes the target software in real-time, exposing flaws that may only manifest under specific runtime conditions. This approach is particularly valuable for uncovering edge cases that are challenging to identify using conventional testing methods.

The **utility** of fuzzing lies in its ability to:

- **Expose Hidden Vulnerabilities**: By generating a vast array of test cases, fuzzing uncovers memory corruption issues (e.g., buffer overflows, use-after-free errors), logic errors, and unhandled exceptions.
- **Automate Testing**: Fuzzers can autonomously produce inputs and monitor program behavior, reducing manual effort while increasing the scope of testing.
- **Improve Software Robustness**: Fuzzing identifies weak points in software, allowing developers to patch vulnerabilities before deployment.

Coverage-guided fuzzing (CGF) is a specialized fuzzing technique that uses **feedback from the execution** of the target program to guide the generation of inputs. Unlike traditional fuzzing, which generates inputs randomly, CGF evaluates the program's code **coverage** to maximize testing efficiency. By analyzing which parts of the code are executed by each input, the fuzzer prioritizes generating inputs that explore untested paths. The key features of coverage-guided fuzzing are instrumentation, a feedback loop and optimization. The first aims at instrumenting the program in compile time so that it can dynamically trace the coverage and keep a record of other useful information. The feedback loop iteratively generates new inputs based on coverage data and uses a heuristic so that it increases coverage, therefore explores more parts of the code and potentially discover new bugs. Finally, fuzzers use an optimization method which targets on reducing redundancy in testing and increase the likelihood of

discovering more complex execution paths. Some famous tools such as AFL++ [46] and libfuzzer are using coverage based fuzzing algorithms.

In this work, we focus on using AFL++ [46] because of its **flexibility** due to the fact that it is a **general purpose fuzzer** meaning that the user can adjust the fuzzing pipeline to inject input in almost any software. It also supports dynamic hooking and support for different platforms and uses plugins on gcc and clang compilers to instrument the code. AFL++ facilitates a variety of fuzzing techniques, as well as multiple running fuzzers which increases the effectiveness of our experiments.

More specifically, AFL++ has support for multiple **sanitizers** with the most famous being Address sanitizer (ASAN), Undefined Behavior Sanitizer (UBSAN) and Thread Sanitizer (TSAN) [46]. Sanitizers are run-time tools that are used for extending the program's capability of detecting illegal behavior, e.g. invalid memory accesses, uninitialized pointers, etc, and dropping execution safely. ASAN finds memory corruption vulnerabilities (i.e. use-after-free, NULL pointer dereference etc.). Following, UBSAN focuses on discovering undefined behavior in C or C++ binaries (i.e. integer overflows, use of uninitialized values, etc.), while TSAN tries to investigate thread race conditions. They can be used during the instrumentation phase by setting the appropriate framework defined environment variables.

The process of fuzzing a binary with AFL++ consists of the following phases

1. **Instrumentation**: Users, compile the source code using different sanitizer options or the default ones. The default and proposed AFL++ sanitizer is *afl-clang-fast* [46], but the user can also add their own clang/gcc compiler flags in case they want to ensure that some optimizations are not applied by the compiler. This might be useful when trying to incrase coverage.
2. **Input Gathering**: Next step is running the program with some input and constructing a corpus, the starting corpus at the beginning of the fuzzing campaign run. It is important that this does not take too much to process and it does not crash the program in any particular way.
3. **Setting up fuzzers**: A user can decide to run a single fuzzer or use multiple cores to run a Main and many secondary fuzzers that are cooperating with each other and share information with the top level one once in a while. Good practices while using multiple fuzzers are to run with different sanitizers and coverage approaches where the focus of each fuzzer is either different or diverges a bit so that the coverage and number of execution paths is maximized.
4. **Running**: When everything is ready we use a dedicated core for every running fuzzer and run the program with the starting input. AFL++ makes sure to mutate the input properly and records coverage and crashes for later investigation. According to the state-of-art-standards the campaign should last at least for 24-hours [36]
5. **Crash Triage and Coverage Inspection**: After the pass of 24-hours, we inspect the crashes for duplicates and false positives using afl-provided tools and manual inspection. At the same time, there is tools such as lcov [23] and gcov [16] that are might help to get a more detailed report of the exact lines that were covered by the mutated inputs.
6. **PoC Construction**: Finally, when the user has a corpus of crashing inputs, they use ASAN [1], gdb [17], valgrind [49] and other tools to further investigate the type of the bug and try to create a Proof-Of-Concept input that displays how the discovered vulnerability can be discovered.
7. After finishing this process the discovered input can be filtered and re-fed to the program for a more fine-grained fuzz campaign, so that the second step can be skipped and the fuzzer can start with higher coverage nor spending time on rediscovering the same paths.

3

Threat Model and CVSS for SpaceOS

This section systematically defines the threat landscape of on-board software in mixed-criticality space applications. We begin by outlining the security assets that must be protected, followed by a discussion of the actors involved, the assumed capabilities of the attacker and the platform, and the key attack vectors that threaten space systems. Examining these elements facilitates as a foundation for understanding the security challenges inherent in mixed-criticality applications. Furthermore, a robust threat model acts like a standardized directive with respect to the suitability of an OS for space missions.

In addition, this section defines a robust risk assessment framework that evaluates the importance of each asset within the mission's operational context. The framework assesses potential vulnerabilities and their impacts while proposing remediation strategies aligned with established space security and safety standards. By integrating this standardized approach, the framework not only supports vulnerability mitigation, but also establishes a consistent methodology for future testing and validation of SpaceOS. This ensures that security practices remain adaptable and relevant to evolving threats in space system deployments.

Although significant efforts have been made to standardize security testing for satellites, highlighting use cases through attack trees [8, 14] and component-based security analysis [13, 21, 53], there remains a critical gap in addressing the unique security challenges of SpaceOS components that are running on-board from an attacker who has already established a foothold on the machine using external attack vectors, such as vulnerabilities in communication protocol implementations or a supply chain attack [54]. Existing security models often focus on initial foothold vectors (e.g., supply chain attacks, protocol vulnerabilities) but fail to explore how an attacker escalates privileges or laterally moves through the onboard software stack post-compromise. This attack model maintains focus on assisting the exploration of the various post-exploitation paths which follow the foothold phase of an attack to a space system.

The motivation behind developing this framework is to bridge this gap by integrating public knowledge on testing RTOS software [4, 20, 38, 41, 47], creating a holistic and structured guide to assess the security of OBSW. Such a framework is essential not only to ensure consistent and robust testing practices but also to underline the operational significance of SpaceOS within the broader mission context, thereby enabling a more targeted and impactful security analysis for future space missions.

3.1. Motivation

Since we aim to create a standardized threat model as close as possible to state-of-the-art space software development frameworks, we consult ECSS and SAVOIR directives regarding OBSW operations. Furthermore, we consider the terrestrial use of general purpose OSes which enable multi-tenancy, given that we aim to explore the behavior of the system where different tasks are interpreted as different actors.

According to ECSS [11] and SAVOIR [35] guidelines, On-Board Software needs to exhibit high reliability as it manages mission critical operations and employ fault containment mechanisms which are essential

for system integrity in the event of a security breach or malfunction. In addition, SAVOIR compliance indicates standardization and modularization of OBSW for facilitating both seamless integration in different systems and isolation of mission-critical functions and data. ECSS compliance enforces policies about resource utilization and satisfaction of real time restrictions according to task criticality for the mission. It also defines least-know, least privilege as general security policies which should be followed during development of space-targeted software [11]. The aforementioned directives are interpreted accordingly for security in the following sections where asset and attacker assumptions are defined.

In order to create a more complete threat model we take inspiration from the threat models of multi-tenant OSes where processes have different privilege levels. Though the operating systems we tested are not what would typically assumed to be multi-tenant, we interpret the "tenant" term by directly binding it to a "task" that is running and the Kernel API of the OS would serve as the centralized "server"-like component. The assets of this system are consisted by memory and process isolation, resource availability and control over privileged operations and functions. The threat actors are either insiders of the system, which in our case would be a task that went rogue, or outsiders, e.g. network attacker that exploits vulnerabilities in other modules to propagate payloads in the system level. The usual attack techniques in this context include DoS attacks, memory or IPC attacks or attack vectors that aim to bypass weak access control mechanisms and execute otherwise privileged functions. In space systems, the multi-tenant system threats are exaggerated due to the highly dependable nature of the system which is grounded in its reliability and resource availability constraints.

3.2. Assets & Security Requirements

Our research focuses on space software utilized in various use cases, including earth observation, satellite navigation, satellite telecommunications, and deep space missions. The nature of the target software and the specific mission requirements determine the critical assets and their associated security requirements.

- **Confidentiality, Integrity and Availability**: Ensuring the security of on-board software requires strict enforcement of confidentiality, integrity, and availability (CIA) principles. Confidentiality protects sensitive information such as mission-specific algorithms and telemetry data from unauthorized access, extending to inter-process communication and shared storage to prevent unintended data exposure. Integrity safeguards critical operations, including attitude control and system communications, by protecting the kernel, configuration files, and task execution from unauthorized tampering, ensuring mission stability and preventing cascading failures. Additionally, data integrity ensures the accuracy of telemetry and control commands through mechanisms like checksums, cryptographic signatures, and redundancy strategies. Availability guarantees continuous operation of real-time tasks and shared resources by protecting critical memory regions and the scheduler from denial-of-service attacks or resource exhaustion, preventing mission disruptions or irreversible damage. Implementing fault-tolerant scheduling, priority enforcement, and resource management policies helps maintain real-time constraints and system resilience under potential threats.
- **Privileged Operations**: Privileged instructions, such as those related to memory management or system state changes, must be protected to prevent unauthorized access and exploitation. These instructions should only be executable by high-privilege tasks or routines, with strict access controls in place to prohibit unauthorized user-space applications from executing privileged operations.
- **Inter-Process Isolation**: This involves ensuring that processes running within the RTOS are securely isolated to prevent data leakage or interference between mixed-criticality applications. Robust isolation mechanisms are necessary to uphold both the integrity and confidentiality of critical processes, especially in systems where high-criticality and low-criticality tasks coexist.
- **Maintain Control**: This involves guaranteeing that a compromise cannot result in permanent loss of control over the system by the legitimate control station (e.g. the Ground Segment). Even in the presence of partial system compromises, mechanisms must be in place to enable the ground station to reassert authority and ensure mission continuity.
- **Timing Guarantees**: Adhering to strict task execution deadlines is essential in real-time operating environments, particularly for high-criticality tasks such as attitude control, propulsion adjustments, and payload operations. Mixed-criticality systems must ensure that high-priority tasks

are completed on time without being interrupted or delayed by lower-priority operations. Timing guarantees also ensure synchronized operation with external systems, such as communication windows with ground stations or data collection events during orbital passes. Real-time schedulers, priority inversion prevention mechanisms, and deterministic execution models are key to achieving these guarantees.

3.3. Actors

In the on-board software execution environment, we can define the following actors, sorted by privilege:

- **Privileged Tasks**, defined as tasks that perform critical operations and should have higher execution priority since their real time constraints are considered to be hard and mission-critical. Their read/write access to the system memory could either be constrained according to policy and they have rights to execute privileged system-calls.
- **Un-privileged Tasks**, which are defined as all of the tasks that should not be treated as mission-critical by the system.

With respect to the system level on which we perform our security testing, we define different level of trust for each one of those actors. When evaluating in the (RT)OS level, we consider that anything hypervisor related in trusted, the kernel is also trusted, high privilege tasks are trusted since they should only concern the mission critical operations and need not interact with other modules but could interact with lower privilege tasks through some IPC mechanisms under well defined access control policies. Finally, low privilege tasks are considered semi-trusted since they should operate under resource restrictions and have limited capability of executing system-calls. Other parts of the system such as the Scheduler, Interrupt Service Routines (ISR) or Peripheral drivers are considered trusted components.

Table 3.1 display some indicative task groups for the platform and payload of a satellite, the resources they use, whether they are mission specific and indicate of whether they should be privileged or not. We elaborate on the privilege based on three distinct characteristics. First, we consider the resources that the task needs to access in order to operate, the more resources the higher the privilege needed for a task. Following, we deem that mission specific tasks, such as science experiment scripts, or tasks running on the payload and handle user input should be considered of low privilege and be sandboxed properly using access control mechanisms and privileged API debloating techniques. Finally, we should consider the impact of a task if it compromised and assess the risk accordingly given the resource access level of the former. For example tasks on the OBC are usually more privileged since they have less interaction with user controlled input compared to the Payload tasks. Nevertheless, consider the case that an attacker manages to upload a software update and manipulate a bug in the Software Update Management logic to gain code execution. In this case, if this OBC task is considered privileged and trusted, thus has extensive access to resources and its enforced access control policy is quite permissive, the attacker can temper with privileged behavior and perform a large impact attack since the OBC is the "heart" of the satellite operations.

3.4. Assumptions

The primary assumption in this threat model is that the attacker has already gained access to a low-privilege (restricted, or semi-privileged) task within the system, possibly through a supply chain attack or by exploiting code execution vulnerabilities in another component that executes in the task environment. Payload-handling tasks, due to their interactions with external data sources and frequent updates, are considered semi-trusted by the system, meaning that their system resource exposure (e.g., memory, system calls, and IPC mechanisms) is restricted. However, due to the shared environment in mixed-criticality systems, their compromise could serve as an entry point for lateral movement.

The attacker is not an external network adversary but operates within the satellite's software ecosystem, potentially possessing partial or extensive knowledge of the satellite system, including its software stack, onboard hardware, and system architecture. It is assumed that direct tampering with satellite hardware is not feasible, and the attacker relies solely on software-level exploitation.

The OS may implement an MPU or MMU to enforce memory protection and might incorporate hardware or software based isolation techniques to differentiate privilege levels. However, these mechanisms may

Name	Description	Resources Used	Mission Specific	Privileged
ADCS Tasks	Maintains satellite orientation and performs attitude corrections using sensors and actuators.	IMU, reaction wheels, star trackers, control buses		✓
Software Update Handler	Handles firmware patching and system updates from ground station.	Bootloader memory, flash memory	✓	
EPS Management	Manages power distribution, battery charging, and solar input regulation.	Battery bus, solar arrays, power switches		✓
TC/TM Tasks	Command Decoder and Executor. Receives and authenticates telecommands and performs critical onboard actions.	Memory management, command queue, system calls		✓
FDIR Tasks	Detects faults in hardware/software and initiates recovery.	System health monitor, watchdogs, reset control		✓
Payload Data Processing	Processes raw payload data such as images or measurements from sensors.	Payload buffers, temporary memory, I/O buses	✓	
Housekeeping Health and Telemetry Aggregator	Collects routine sensor data (temperature, voltage, status) for transmission.	Timers, ADC channels, memory buffer		
Data Compression Task	Compresses data before storage or downlink to ground station.	RAM, CPU time, compression buffer, Third Party Libraries.	✓	
Science Experiment Scripts	Custom software handling mission-specific experiments (e.g., CubeSat scientific payloads).	Data logging, file system, custom APIs	✓	
File Transfer and Logging	Moves internal files, prepares downlink packets, logs telemetry.	File system access, storage medium		

Table 3.1: Example of privileged and unprivileged onboard software tasks in a mixed-criticality satellite system. The existence and categorization of such tasks groups was infered from SAVOIR [35] and ECSS documentation [10, 11, 22, 42, 48]

not be fully configured or may contain implementation flaws that can be leveraged by the attacker.

Once a foothold is established in a low-privilege task or application, the attacker's primary objective is privilege escalation. This may be achieved through inter-process communication (IPC) vulnerabilities, system calls, weak access control policies, or memory corruption exploits. Additionally, attackers might aim to manipulate the real-time scheduler, leading to task starvation, priority inversion, or denial-of-service (DoS) conditions that disrupt critical satellite functions.

This threat model assumes that cryptographic protections are implemented and cannot be trivially bypassed. Physical-level attacks such as fault injection or side-channel analysis are considered outside the scope of this work.

3.5. Threat Vectors & Attack Surface

This section details the primary threat vectors and attack surfaces relevant to Real-Time Operating Systems (RTOS) in mixed-criticality satellite systems. Given our assumption that the attacker has already gained access to a process within the system, the focus is on vectors that could lead to privilege escalation, compromise system integrity, or bypass critical isolation mechanisms.

RTOS-based systems in satellites are primarily responsible for managing real-time operations, including high-priority tasks such as attitude control, communication handling, and other mission-critical functions. The RTOS attack surface includes process memory, system calls, IPC channels, and scheduling mechanisms, each of which could be exploited to compromise system integrity, availability, or control.

- **Memory Management Exploits**: Memory corruption attacks, such as buffer overflows, stack smashing, and use-after-free vulnerabilities, are primary threats to RTOS security. These attacks can allow an attacker to alter system memory, leading to arbitrary code execution, privilege escalation, or data leakage. Since RTOS systems lack memory protection found in modern OS, this vector is critical for compromising memory-sensitive assets like the kernel or scheduler.
- **Inter-Process Communication Exploits**: Given the mixed-criticality nature of satellite systems, IPC is crucial for communication between high and low-criticality tasks. Attacks on IPC mechanisms, such as message tampering or unauthorized data interception, can lead to interference with critical operations or bypassing of process isolation. Compromising IPC could impact both Inter-Process Isolation and System Integrity by allowing an attacker to manipulate data or timing between tasks.
- **Insecure Syscalls**: System calls are often a key point of attack in RTOS environments. Insecure or improperly validated system calls can enable low-privilege tasks to escalate privileges or gain unauthorized access to critical functions. Since syscalls provide direct access to kernel-level functions, this vector could compromise Privileged Operations and potentially lead to system-wide impact.
- **Scheduler Attacks**: The RTOS scheduler is responsible for ensuring that high-priority, time-critical tasks execute as required. Scheduler attacks, such as priority inversion or denial-of-service attacks, can prevent critical tasks from meeting real-time deadlines, which is particularly dangerous in satellite systems where missed deadlines can have physical impacts. This vector is closely tied to System Availability and Maintain Control requirements, as compromising the scheduler can lead to severe mission failures.
- **Boot Memory Compromise**: Compromising the RTOS during its boot process, particularly by targeting the memory where initial configurations or the kernel are loaded, can allow an attacker to install persistent malicious code. This threat vector directly impacts System Integrity and Control Maintenance, as it could prevent the ground station from regaining control over a compromised system.
- **Software Design Issues**: Design flaws within the RTOS, such as inadequate memory segmentation or weak access control policies, create security gaps that attackers can exploit to bypass isolation mechanisms or escalate privileges. These issues are critical in mixed-criticality environments, where the failure to isolate low- and high-criticality processes could compromise System Integrity and Guest Application Isolation by allowing lower-priority processes to interfere with or access sensitive functions.

3.6. CVSS for SpaceOS

CVSS [9] is a useful tool for a security expert to assess the severity of vulnerabilities and it is fitting for findings coming from application security testing and red teaming operations. Nevertheless, to map the metrics on the space system requirements based on mixed criticality systems we need some further refinement. In this section, we adjust the interpretation of the existing metrics of CVSS v3 so that space developers can use the comprehensive framework to assess severity of findings in OBC software. The modifications are made with the space segment in mind to fit our research, while they can similarly be used in Ground and User Segment testing operations.

3.6.1. Base Metrics

Attack Vector (AV) Attack Vector exploitability metric reflects the context of the exploitation in terms on the character of the attackers' operation in order to exploit the vulnerability. The default version has 4 different scores, namely Network, Adjacent, Local and Physical. We provide a new interpretation for each one of them. More specifically, Network attack vectors do not bound the attacker in a strict way, meaning that they could execute the attack even from the Ground or User Segments and they only require network access from the attacker. Adjacent attack vectors require that the attacker has compromised a component in the Space Segment, such as another payload or platform, which can communicate with the target system through some space communication protocol, e.g. CSP, or an avionics bus, e.g. CAN-Bus. An example of an adjacent attack would be an attacker controlling some application and trying to compromise a different payload that has established communication with the compromised domain. Local attacks are more restricting since they require the attacker to be able to perform arbitrary read/write/execute on the same application/component domain as the target. In some cases, the attacker might utilize debugging shells or a file loading capability to achieve such context. Finally, physical attacks require the attacker to has physical access to the device, which would be the case for attacks that take place before deploying the satellite. This type of attack vectors are more rare but could happend when the threat vector is a firmware or hardware attack that needs physical loading of the exploit in the target component.

Attack Complexity For the Attack complexity metric we keep the default metrics, i.e. Low and High. Low complexity attacks do not require specialized circumstances to take place, while High complexity exploitation depends on execution environment and target-specific reconnaissance. Extensive preparation and attack path analysis is necessary on the attacker end for such an attack to succeed in a real target environment. The latter should be the case in the mixed criticality systems domain and even more often in the space OS context, since complex system-design and mission-specific platforms make each sub-system and component of the OBC special.

Privileges Required For Privilege metric we are extending the interpretation of the generic triplet in the default version of the framework (None, Low, High) to a fit our use case. No privilege would mean that the exploitation of the threat vector is independent of the privilege level in the execution environment this takes place. Low privilege could be fit in the example of a low-privileged, or restricted task (e.g. no access to dangerous syscalls, or even forbiden to read/write/execute some parts of the memory) that goes rogue and tries to compromise other peer or privileged tasks. A high privilege could be interpreted as the case of task that performs dangerous syscalls and has access to sensitive information, i.e. configuration files or OS-sensitive state objects in memory, which could critically impact the system state.

User Interaction In the default version, the User Interaction metric captures the requirements of a benign User participating for the attack to be successful. For space systems, though the interaction of a GS operator might be needed for an issue to be exploited, we extend the meaning of a "user" interaction to "component" interaction since IPC and periodic operations are common on satellite systems, enabling a redefinition of "who" we are trying to attack. It would make sense in such a system to await for a ping message or a specifically timed interruption to occur before initiating an attack. Later on we will understand why timing might be an important factor to attack OBSW.

Scope The Scope metric has two possible scores, "Unchanged" and "Changed". Though we are suggesting that the boundaries between different domains and scopes should be defined accordingly in the beginning of a security test, we give an example mapping to our use case. Given an attack vector, our scope is unchanged when the vulnerability does not compromise a different execution context or memory domain that is separated by the current domain during the design phase. For instance, an application running in a payload which is the target of an attacker that controls and abuse the capabilities of another application that runs in a neighboring payload. Similarly to this scope notion, we make our assumptions for memory domains and address spaces, e.g. threads are running in the same address space, meaning that compromising a task from another different task under the same execution context is not considered a scope-change. This details are specific to the system that is tested and to the character of the assessment, aligning with the notion that vectors aiming at privilege escalation are usually indicating a change of scope after successful exploitation.

Impact Metrics (CIA) The Impact Metrics, i.e. Confidentiality, Integrity and Availability, are global and well defined in any context so there is no need to deviate from or redefine them in our testing scope, therefore we continue with the default definition [9].

3.6.2. Temporal Metrics

Exploit Code Maturity We are keeping the same metrics as in the default CVSS v3 [9], i.e. Not Defined, High, Functional, Proof-Of-Concept and Unproved, with some small changes on the definitions. While the Not Defined and Unproven definitions stay the same, the other three must be adjusted to the MCS and Space OS context that makes the testing more difficult due to the different stages of a mission in which a vulnerability might be discovered. Proof Of Concept and Functional attack vectors refer to exploit code that works on a simulation or a similar testbed but not on the real system. Highly mature exploits are tested in the actual mission hardware and are for sure posing a threat to the mission. For the later case to be meaningful, it is necessary to test the exploit code before the deployment stage of the mission.

Remediation Level Remediation Level of a vulnerability refers to the state of a mitigation patch or a workaround published by the vendor that provides the affected component. Since this is not a special case in the space industry we are keeping the default definitions [9]

Report Confidence Report Confidence is also affected in the space context. Similarly to the exploit code maturity metric, the different development and integration phases of a space mission affect the confidence as to if a vulnerability will exist in the final product. Therefore we are mapping the "Reasonable" and "Confirmed" scores [9] to **"Simulation Tested"** and **"HW Tested"** to represent the idiosyncrasies of space OS security testing. A confidence scored as "Simulation Tested" means that our exploit works in a simulation of the real system where parts or the totality of the hardware and firmware is emulated for testing and affordability purposes. This could mean that the tests are performed in a ground-based testbed with partially representative hardware. "HW Tested" confidence means that all of the hardware and software parts are exactly the same as used in the deployment version of the space segment component under attack. This means that the development cycle is done and we are one step before the actual deployment. This distinction happens due to the nature of using COTS in space systems where small tweaks might happen throughout the development which might cause the exploit tested in a simulation of the environment to fail in any way.

3.6.3. Environmental Metrics

Environmental Metrics consist of Security Requirements with respect to the CIA features of the protected assets and the Modified Base Metrics. The security requirements are different for each distinct use case and are difficult to standardize since **they are mission and organization specific**. We keep the generalized version as shown in the CVSS v3 specification [9] defining 4 different scales, i.e. Not Defined, High, Medium, Low. The Modifies Base Metrics are again remaining the same. This metric captures the changes which the testing environment has to the Base Metric Scores. For example if we are performing tests under a simulated environment with any authentication or authorization checks turned off, the environmental scores for "Privilege Required" and "Attack Vector" should change. An-

other example would be attacks that exploit vulnerabilities in the TC/TM traffic parsers which require authentication/authorization and maybe a firewall is set between the target and the system. If the assessment requires to disable these defenses then the environmental scores should be adjusted properly during vulnerability scoring.

3.6.4. Disclaimer for RTOS

At this point, let us make a note that typically Real-Time Systems are usually not created with security in mind as a top priority. This research explores this aspect by performing an implementation and design review of the vanilla FreeRTOS kernel. Therefore, the discovered issues elaborate on why these types of OS's are not suitable when considering the security of the OBSW in a real mission environment. Nevertheless, more safe and secure options such as LynxOS or VxWorks are trading off performance to achieve access control and better inter-task isolation. Since to our knowledge from open source projects, systems such as FreeRTOS and RTEMS are widely used in space missions, we proceed to apply our assessment on the former and evaluate the whole framework, from the automated vulnerability discovery to the point of vulnerability assessment and impact scoring.

4

Space-OS Security Experiment Scenarios

In this chapter, we discuss the requirements for setting up a successful security lab to perform security tests on space OS. Our approach explores different use cases that a developer could test on a satellite testbed. For each one, we go over the necessary components needed during the testing phase, touching both the user-end and the testbed requirements. We suggest guidelines to test for the different threat vectors discussed in Section 3.5 and give a general description of the given experiment and what its goal would be. In Table 4.1 and Table 4.2, we display the technical requirements of the distinct experiments for deployment in both a simulated and a representative environment (i.e. ground and space segment are present and distinct), which we deduced during our experimental work. Though in most use-cases the experiments could also take place in a single machine using simulators, it is important to consider the technical components necessary to execute an experiment in space-representative conditions.

4.1. OBSW Attacks

These experiments are focused on compromising the on-board components, mainly focused on the software assets of the platform.

4.1.1. Denial Of Service

This experiment simulates a rogue software piece or process that disrupts the availability of other processes of the same or different criticality. The main objective of the attacker is to compromise the availability of the systems resources and make critical tasks fail. The experiments goal is to test the system's ability to prevent, detect and recover from resource exhausting or task starvation attacks. Looking for threat vectors such as "Scheduler Attacks" and "Software Design Issues" could be used to guide the testing phase.

4.1.2. Privilege Escalation

A user could test different vectors such as memory management, system calls or inter-process communication API's to find vulnerabilities and compromise higher-privilege assets without authorization. The goal of this experiment evaluates the resilience of the underlying RTOS with respect to privilege escalation attacks that could lead to a complete seizure of control of the undelying component. Discovering threat vectors in the types of "Software Design Issues", "IPC Exploits", "Memory Mangement Exploits", "Malicious Updates and 3rd-Party SW", or "Insecure Syscalls/Hypercalls" could result in a privilege escalation vector.

4.1.3. Dangerous System Call Monitoring/Prevention

Attackers main way of meaningfully exploit a process is by having access to syscalls. Invoking dangerous syscalls that could cause confidentiality compromise of sensitive data, to arbitrary code execution on

Requirement	Description
Computing Workstation	High-performance CPUs, robust memory, enough RAM to execute multiple heavy-{processing, memory} processes for log saving, fuzzing, network operations, etc.
Virtualization SW	Robust and tested virtualization software to simulate different architectures.
Cross-Compiling Tool-Chains	In order to simulate and debug in different architectures it is usually essential to compile before running an app on a simulator or virtualization software.
Exploit Development Tool-Chain	For creating proof-of-concept exploits and automate the exploitation phase.
Remote Access Software	To connect to different virtualized machines and fast-track the debugging experience.
Firmware and System Level Debugging Tools	For low-level debugging.
Command and Control Software	To simulate on-the-fly update process.
Protocol Simulators	In some cases, for communicating with different software components that use a specific communication protocol.

Table 4.1: Technical requirements for the "Simulator" section, the experimental part of the lab which focuses in automating the assessment pipeline and is usually low-budget, based on mainstream hardware and quite flexible in terms of set-up.

the component. The goal of this experiment is to test the RTOS syscall monitoring capabilities and discover if they give excessive syscall permissions to the different applications. Threat vectors such as "Insecure Syscall" and "Software Design Issues" could be used to test the robustness of this feature.

4.1.4. Scheduler Exploitation

Schedulers in RTOS can be evaluated for vulnerabilities such as task priority inversion, resource starvation, and timing attacks. This experiment aims to assess the scheduler's resilience and the impact of these attacks on high-criticality, time-sensitive tasks. Additionally, it investigates RTOS policies regarding access control for tasks with varying criticality and privilege levels, focusing on their interactions with the scheduler. The identified "Scheduler Attacks" vectors and potential "Software Design" flaws provide a foundation for developing proof-of-concept experiments to validate these findings during the testing phase.

4.2. OBSW Fuzzing

These experiments focus on fuzzing techniques to uncover vulnerabilities in on-board software (OBSW), including RTOS kernels and communication protocols' libraries. The goal is to evaluate the robustness of these components against malformed inputs and unexpected operational conditions.

4.2.1. RTOS Kernel Fuzzing

In this experiment, the user tests the RTOS kernel using state-aware fuzzing techniques to identify vulnerabilities in critical components such as system calls, memory management, and task scheduling. The goal is to assess the resilience of the RTOS kernel to malformed or malicious inputs and uncover potential weaknesses that could compromise system confidentiality, integrity or availability. Targets threat vectors such as "Memory Management Exploits" or "Insecure Syscalls".

Ground Segment Requirements

Requirement	Description
Workstation	Base station for communicating with the space segment.
Cross-Compiling Tool-Chains	Compiles binaries for different architectures before flashing them to the OBC.
Exploit Development Tool-Chain	Used for creating and propagating exploits, automating the exploitation phase.
Remote Access Software	Enables connection to remote or local interfaces, improving the OBSW debugging experience.
Firmware and System-Level Debugging Tools	Facilitates low-level debugging of onboard firmware.
Command and Control Software	Enables on-the-fly updates of the Space Segment.
Network Interfaces and Hardware	Equipment and software/firmware for communication between GS and the Space Segment, e.g., CAN-Bus port or antenna for sending TCs.

Space Segment Requirements

Requirement	Description
On-Board Computer (OBC)	Configured hardware supporting the targeted software platforms. Can include hardware security features, multiple architectures, and additional modules for system reliability testing.
On-Board Software (OBSW)	Includes hypervisors, Linux Systems, RTOS, and necessary drivers ported to hardware and flashed onto the OBC. Software must be configured for mission-specific requirements.
Electrical Power Supply (EPS)	Supports the power requirements of the space segment, ensuring power generation and distribution for both flat-sat and airborne satellite setups.
Communications Module (COM)	Parses and propagates TC/TM traffic to relevant components.
CAN-Bus	Facilitates communication between onboard components and provides connectivity to the ground station in a flat-sat setup.

Table 4.2: Requirements for testing vulnerabilities in a representative environment. The Ground Segment and Space Segment components reflect real-world platforms used in space missions and are configured as close as possible to their final mission deployment.

4.2.2. Protocol Fuzzing

In this experiment, the users fuzz communication protocol libraries used in software development for inter-process communication (IPC) or inter-component interactions, injecting malformed or invalid data

to evaluate the systems response. The goal is to test the robustness of communication protocols and their implementation against exploitation, ensuring secure and reliable data transmission between processes or components. The threat vectors that could be related to this experiment are "Software Design Issues", "IPC Exploits" and "Memory Management Exploits".

4.3. On-Board Systems Isolation

These experiments assess the robustness of isolation mechanisms at various system levels.

4.3.1. Task/Process Isolation

Mainly concerning the RTOS level, a user can experiment with inter-process access rights mechanisms and explore the memory and scheduler isolation policies, if any.

Memory Isolation

In this experiment, a user test for unauthorized access to memory areas assigned to other tasks or processes. The goal is to validate the underlying segregation mechanisms for ensuring that critical memory areas remain inaccessible to unauthorized entities.

Scheduler Isolation

A compromised task could try to tamper with the priority of other running tasks and according to the criticality and urgency of the target task make the system fail and even permanently damage the mission. The goal of the experiment is to test the Scheduler integrity policy, enforced by the RTOS.

4.3.2. Mitigation Enhancements Performance Measurement

As a defense focused experiment, users could apply known segregation techniques and other memory corruption mitigations and run previously working Proof-Of-Concept test-suites to ensure the security of the mitigation. At the same time, the user would run bench-mark programs and compare the results with the original set-up to report the trade-off between security and performance, which is a crucial element in the use case of space missions.

4.4. Secure Firmware

These experiments focus on testing the boot process to ensure the integrity of the system from startup. The goal is to evaluate the robustness of the secure boot mechanism when malicious and unauthenticated software/firmware changes happen.

4.4.1. Secure Boot

In this experiment, we test the secure bootloader by supplying insecure and modified images and try to boot into them. The goal of the experiment is to verify that the untrusted images are forbidden while the bootloader would raise an error in case of unsigned and unknown images. The experiment is tightly related to "Software Design Issues" regarding the bootloader firmware used by the hypervisor.

4.4.2. Boot-Time Memory Corruption

The user could target the memory used during the boot process to test for vulnerabilities in kernel initialization that could result in attacker persistence issues if attacked. The goal of this experiment is to ensure the bootloader and kernel are resistant to persistent malicious code injection. Threat vectors that could be applied in this experiment are classified as "Boot Memory Compromise" vectors.

4.4.3. Supply Chain Attack - Malicious Firmware Detection

Users test the secure boot and image signature checking mechanisms for covert supply-chain attacks, trying to trick the bootloader into loading into maliciously updated firmware. The goal of this experiment is to ensure that the bootload will reject such updates and raise an error. Vectors from the "Malicious Updates and 3rd-Party SW" type could be used in this experiment.

5

Results and Evaluation

In this section, we present our experimental work on RTOS security and the dynamic testing toolchain we create in order to fuzz and crash triage the vanilla version of the FreeRTOS Kernel. We tested 4 different versions of the kernel, as displayed in Table 5.1, and discovered 7 potential security issues.

First, we present the technical setup for the simulator in which we run the fuzz campaigns and the representative hardware set-up which was used to confirm the applicability of the issues in a space-like set-up. Then, we go over our methodology, explain the different components of the framework and showcase it's effectiveness for the different kernel versions we fuzzed.

5.1. Set-Up

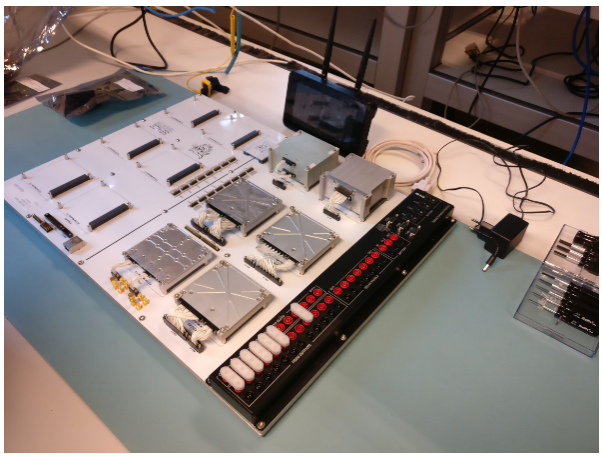

Figure 5.1: The Cube-FlatSAT platform.

We implemented our main fuzzing actions using `AFL++` and the vanilla FreeRTOS Kernel version. The fuzzing campaigns were run in a Virtual Machine (VM) using the POSIX Simulation for the FreeRTOS Kernel. We implemented a compatibility layer to invoke the syscalls and C scripts to generate the initial input corpus and parse the crashing input to a human-readable version. We also implemented coverage and crash triaging tools running in the x86_64 simulation. After discovering some potential issues, we tested their applicability in a representative Cube-FlatSAT environment, as seen in Figure 5.2. Following, we present a comprehensive set-up and dependency guide to use our tool in a x86_64 environment and an overview of the representative Cube-FlatSAT hardware which was used to confirm the findings.

5.1.1. Simulation

As shown in Table 5.1, we are using a VM running Ubuntu 22.04 as the Host OS of our experiments. Therefore, we are using the POSIX simulation port for the FreeRTOS kernel. We are fixing the rest of the dependencies to make sure that there is no variation in how the binaries are compiled and that we are debugging and calculating coverage with the same software for every one of the different RTOS version. Note that for the different RTOS versions we needed to patch some files to support the gcc/clang versions, otherwise some unsigned long warnings turn to errors. Another important note is that afl version should also be kept fixed since we faced some unexplained issues with updates after the specified version. Nevertheless, the pipeline remains the same and by updating the compilers and rest of dependencies appropriately the problems go away but, for sake of simplicity, we keep the `AFL++` version fixed as well.

Table 5.1: List of dependencies and their versions for our experiments on the Simulator

Software Type	Dependency	Version
OS	Ubuntu	22.04
	FreeRTOS kernel	v11.1.0, v11.0.1, v11.0.0, v10.6.2
Misc SW	CMake	v3.22.1
	Make	v4.3
	afl-fuzz	v4.22a
	gcov	v11.4.0
	lcov	v1.14
Compiler	gcc	v4.11.2.0
	clang	v14.0.0
Debugging SW	gdb	v12.1
	valgrind	v3.18.1

5.1.2. FlatSAT Testbed

The components of interest that consist the representative testbed on which we tested our findings are the OBC HW and the OBSW RTOS versions. The OBC module of the flatsat is consisted of two separate OBC modules based on the Microchip SAM-E70 microcontroller. The OBC units have ARM Cortex M7 processors with floating-point and cryptography support which have their own SRAM and are connected to the main bus as shown in Figure 5.2.

On the software end, on top of the HW porting firmware, there is a customizable version of the FreeRTOS kernel, along with a number of plugin libraries, which we will not mention since they are out of scope for this experiment. All applications are running in the User Space, as shown in Figure 5.3, along with watchdog firmware and CSP related services. Since the flatsat OBC is running software on an ARM CPU, we need an arm cross compilation tool-chain [1] and a medium to flash our applications on the OBC.

5.2. Methodology

Given the complexity of RTOS and its monolithic design, static analysis alone is insufficient to uncover security flaws. To address this, we employ fuzz testing, which provides a more dynamic and exploratory approach to vulnerability discovery. In Figure 5.4, we see the basic steps followed in our framework to discover valid crashes. Starting from input generation and harness function development, we try to minimize the manual work needed by crossing out any false positives so that crash inspection is more targeted.

[1] `https://developer.arm.com/Tools%20and%20Software/GNU%20Toolchain`

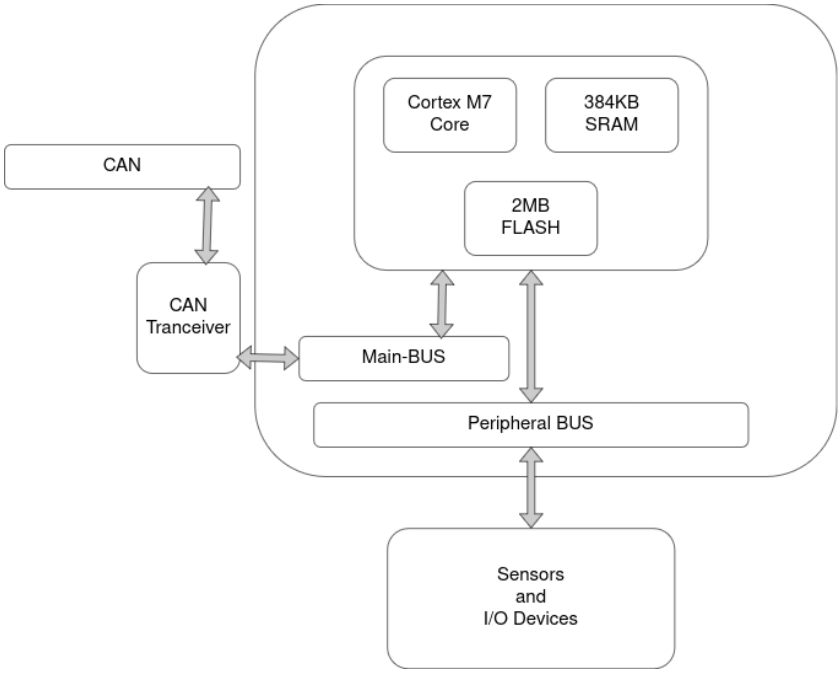

Figure 5.2: Cube-FlatSAT OBC Design. The OBC consists, among other components, from a Cortex-M7 Core, an SRAM, Flash memory and a Main and a Peripheral Bus. Through the main-bus it is capable of communicating over the CAN-Bus with the rest of the OBC units on the platform. We are using the CAN-Bus to flash our images and receive CSP traffic as we will see later.

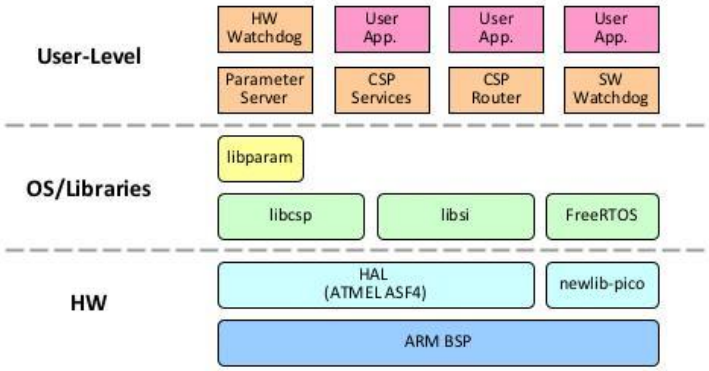

Figure 5.3: The OBSW Design of the Cube-FlatSAT's OBC. Our images will use the OS and library software stack to first build a simulator of the OBC runing on x86 device using the POSIX simulation and after we test it, we cross compile the image, ready to run on the OBC.

Input Generation The first step in fuzzing the kernel is generating the input corpus for `AFL++`. Since `AFL++` is performing mutations in a binary level the starting, we decided to represent a single `AFL++` input as a binary dump of a struct the fields of which are the input parameters for the decided syscalls. We include all of the parameters of the selected syscalls in this struct model and use the I/O methods provided by the C programming language to create the binary dump. Importantly, these inputs should result in a valid program execution that returns zero to `AFL++` fork server since the latter requires a valid program execution as its starting point. After that `AFL++` will generate mutations of this bitstring and on our side we parse it as a struct and invoke the selected syscalls with the altered input parameters. Since it is troublesome to generate valid pointer-like arguments, we chose to include only number or text-based parameters in the bitstring and fix any pointer arguments that the selected syscalls require. Nevertheless, size attributes of this objects could be fuzzed, so we also include such parameters to dynamically generate the required objects. The implementation of this step consists of a C and a bash script that automatically creates a input directory with a single starting input inside it. This is enough

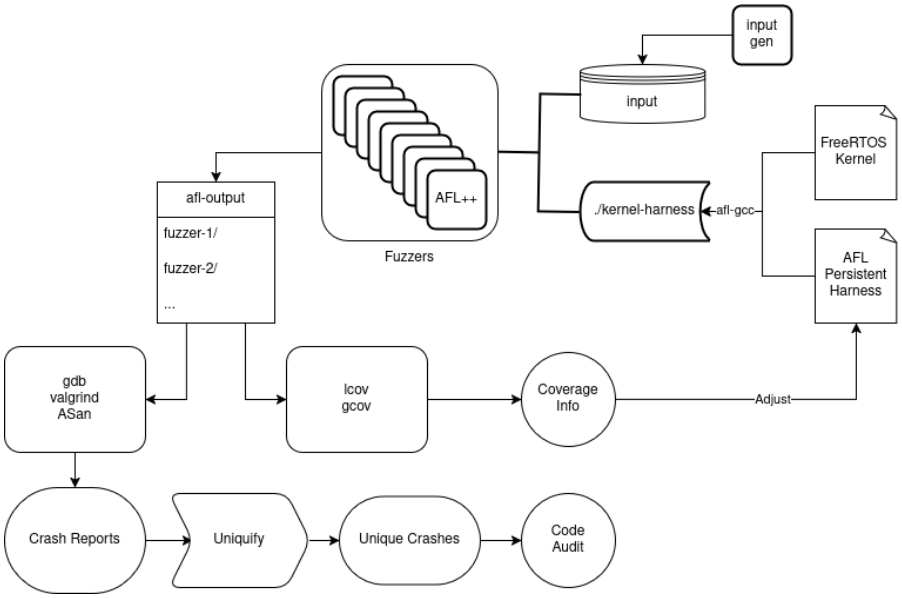

Figure 5.4: The assessment pipeline for our framework. First the developer generates input files and the binary to be fuzzed. Then the fuzzers are run for 24-hours and an output directory with every fuzzers output is generated. Then the developer uses `lcov` and `gcov` to generate coverage information and adjust the harness function accordingly to achieve better code coverage, thus potentially discover more crashes. Then use `gdb`, `valgrind` and ASan enhanced binaries in order to generate crash reports for all discovered crashes. Finally, uniquify them and perform manual inspection of the issues through manual code auditing.

for the fuzzers to start from and it ensures that the initial run of the program does not crash, a behavior necessary for running `AFL++` properly [46].

Kernel Instrumentation The second step in fuzzing the FreeRTOS kernel source code is instrumenting with the afl-gcc-fast compiler. To solve inline function issues and generate a properly instrumented binary we are removing optimizations and add the "`-fno-inline`" flags to the compilation scripts. More specifically, we create a "`CMakeLists.txt`" file that uses the respective Kernel cmake file to compile our harness code along with the kernel source code. Furthermore, `AFL++` provides the user with a choice of instrumenting specific files only. Taking advantage of this feature, we made sure to only instrument the base code files of the kernel, namely "`tasks.c`", "`queue.c`", "`list.c`" and "`stream_buffer.c`". The selection of these files is supported by three arguments. First, we wanted to focus on the logic of the API calls we aim to fuzz, then, the main functionality of the kernel is consisted of these four files, so they guide our fuzzing no matter the instrumentation noise of the other functions, and finally they are remaining relatively stable across different versions, meaning that finding bugs in the logic of the contained methods would potentially affect multiple versions. Additionally, we are also using different sanitizers to run multiple fuzzers and discover different types of crashes. `AFL++` instrumentation process requires to recompile the program with the necessary environmental flags set so that it knows which sanitizer to use for each run. We automate this process by using a bash script that compiles the program, starts the campaign, and then recompiles with different sanitizer enabled, until we are done with all of the cases we want to cover. The coverage displayed by the `AFL++` statistics page is therefore not completely reflective of the actual code coverage over the whole kernel codebase. For that we employ a secondary coverage tool which enables a better view of the total coverage and also visualizes this on the source code folders to better guide the manual code auditing necessary to confirm any potential bugs discovered.

Syscall Selection For our experiments we decided to choose a representative subset of the FreeRTOS kernel syscalls as shown in Table 5.2. The underlying motivation of this decision is coming from extensively analyzing the FreeRTOS Kernel documentation [15] which resulting in the following three selection criteria. First, the amount of numerical, size-like, and text-based parameters of a syscall, which fits the input struct model and enables more effective mutations. Second the behavior of the

syscalls towards memory objects. For example object creation functions that use a static buffer seem to use a memset call based on user-controlled size parameters which is a good indicator of a possible buffer overflow attack, even in a data-only fashion. Finally, we choose some syscalls for each different object and try to cover the basic functionality that would take place in a normal RTOS application with multiple tasks that communicate with each other. This means that, for instance, when testing syscalls that concern communication objects we need to check both send and receive operations. For creating a more representative syscall grouping based on the MCS environment of the Space Segment, we would need a system-call log of a mission so that we can infer the importance and availability of a syscall. Importantly, some syscalls are not provided in all of the versions (older version might miss some as shown in Table 5.2), fact that should be taken under consideration when the overall coverage seems to decline for earlier versions of the kernel.

Harness Functions One of the most important parts of fuzzing the FreeRTOS kernel is defining an effective and flexible harness function that, first, covers the most important functionality of the kernel and, second, fits the restrictions of the input struct. Our framework supports two different versions of the same core. The first is a single-threaded version of the program which sequentially invokes the syscalls and is focused on the overall performance and speed. The second one is a multi-threaded version of the first that utilizes 2 tasks with the same executor function and tries to discover race conditions and other bugs that might occur in an environment with multiple threads running. For this version to work, we first start the two tasks and then execute a monitor task which sole job is to monitor the active tasks, when the number of active non-system tasks is zero then it stops the program and exits execution. The core execution of the harness function consists of parsing the mutated input bitstring, checking the length, passing the parameters in the syscalls and returning zero to the system in case of successful execution. The C code implementing the harness is generic enough to support different fuzzing methods enabling the user to define any structure as the input and any handler function as the parameter passing routine. This way we can glue our framework to any user defined fuzzing methodologies without the latter implementing anything but the fuzzing behavior and defining the input struct.

Fine-Tuning `AFL++`: Coverage vs Performance Besides defining input corpus, syscall fuzzing group and the harness function, it is important to fine tune `AFL++` for performance and better coverage. Since there are different mutation and selection algorithms for guiding the coverage-based algorithm along with multiple choices for integrating sanitizers and discover new bugs, we chose to follow the suggested practise from the `AFL++` docs [46]. First change to our current model would be to adjust the harness to support persistent mode, which enables a more flexible harness function and potentially enhances performance up to x20 percent [46]. At the same time we use the shared memory buffer provided by `AFL++` in order to remove the continuous I/O operations between the input file and the feedback loop of `AFL++`. Furthermore, we incorporate different sanitizers such as Address Sanitizer (ASAN) and Unexpected Behavior Sanitizer (UBSAN) to increase the crashes found by the program. As displayed in Table 5.3, we are also incorporating the CMPLOG [52] algorithm to improve the performance of 3 fuzzers. CMPLOG uses caching and shared memory to tackle the low entropy starting seed state that usually causes `AFL++` fuzzing campaigns to yield low performance in both total coverage and crash discovery speed. In total, 9 `AFL++` fuzzers are deployed for 24 hours [36] in the dedicated VM machine. All of the secondary ones are synchronizing periodically with the main fuzzer and therefore the different mutations of the input that promote larger coverage are shared among the different fuzzers.

Crash Triaging with gdb & valgrind After the fuzzing campaign is done, the next step in to triage the crashes, filter out any false positives and manually inspect the resulting crashing corpus along with the source code in order to discover vulnerabilities in the FreeRTOS kernel API. For this process we created a script that compiles and runs the kernel and functional part of the harness that invokes the syscalls against the user provided input. We are using `gdb` [17] with gef-plugin [7], `valgrind` [49] and Address Sanitizer plug-in for `gcc` [1] to run the programs that were compiled with the `-g3` flag in order to provide a call stack trace and detailed debugging logs. This logs are saved in respective directories and are named accordingly based on the input used. We are currently considering only the input corpus provided by the "crash" and "hang" directories generated by afl. We are running each one of them with the "timeout" command that sends a `SIGKILL` signal to the process after a timeout of 2-3 seconds. If the program crashes before that it is considered a "crash", otherwise it is classified as a hang. Based on

Name	Description
xTaskCreate	Creates a new task and adds it to the ready list. The task will run when scheduled by the RTOS.
xTaskCreateStatic	Creates a task in a statically allocated memory space, removing the need for dynamic allocation.
vTaskDelete	Deletes a task, removing it from the RTOS scheduler and freeing its memory if dynamically allocated.
xQueueCreate	Creates a new queue for inter-task communication, specifying its length and item size.
xQueueCreateStatic	Creates a queue with statically allocated memory, preventing the use of dynamic memory.
xQueueSend	Sends an item to a queue, blocking if necessary depending on the specified timeout.
xQueueReceive	Receives an item from a queue, with optional blocking behavior if the queue is empty.
vQueueAddToRegistry	Adds a queue to the registry for debugging and visualization purposes.
xStreamBufferCreate	Creates a stream buffer, allowing data to be sent and received as a continuous stream.
xStreamBufferCreateStatic	Creates a statically allocated stream buffer, preventing the use of dynamic memory.
xStreamBatchingBufferCreate	Creates a batching stream buffer, optimizing transfers of data chunks.
xStreamBatchingBufferCreateStatic	Creates a statically allocated batching stream buffer.
xStreamBufferSend	Sends data to a stream buffer, blocking if necessary based on space availability.
xStreamBufferReceive	Reads data from a stream buffer, blocking if necessary based on data availability.
xMessageBufferCreate	Creates a message buffer for sending discrete messages between tasks.
xMessageBufferCreateStatic	Creates a statically allocated message buffer, preventing dynamic allocation.
xMessageBufferSend	Sends a message to a message buffer, blocking if necessary based on buffer availability.
xMessageBufferReceive	Receives a message from a message buffer, blocking if necessary if empty.
vTaskStartScheduler	Starts the FreeRTOS Scheduler.

Table 5.2: FreeRTOS Kernel Syscalls selected for fuzzing in our experimentation. All system calls except `xStreamBatchingBufferCreate` and `xStreamBatchingBufferCreateStatic` are available in all four tested FreeRTOS versions. These two syscalls are only available in version `11.1.0`.

the return code of the application we classify crashes and hangs. We noticed that executions returning 139 are resulting in core-dumps, while hangs are returning 124. The rest of the input is stored under the "other" directory and their logs are not saved. Furthermore, we provide the user with a script to uniquify the crashes based on the debugging output and give the option to perform the uniquification based on a specific debugging tool's logs. At this point, a developer must proceed with manual check of the remaining unique true positives and perform targeted manual code auditing aided by this insight in order to infer if the bug is a security issue. The scripts facilitating this procedures are available in

Fuzzer Name	Sanitizer	CMPLOG	Strategy Parameters
`fuzzer-sanitizers-asan-1`	ASan	✓	• `-P explore`: maximize code coverage • `-a binary`: handle binary inputs • `-p exploit`: prioritize triggering crashes
`fuzzer-sanitizers-asan-2`	ASan		• `-P exploit`: focus on triggering crashes • `-p explore`: maximize coverage
`fuzzer-sanitizers-asan-3`	ASan		• `-P explore`: maximize coverage • `-p fast`: favor quick execution paths
`fuzzer-sanitizers-ubsan-1`	UBSan	✓	• `-P explore`: maximize coverage • `-a binary`: handle binary inputs • `-p exploit`: prioritize triggering crashes
`fuzzer-sanitizers-ubsan-2`	UBSan		• `-P exploit`: focus on triggering crashes • `-p explore`: maximize coverage
`fuzzer-sanitizers-ubsan-3`	UBSan		• `-P explore`: maximize coverage • `-p fast`: favor quick execution paths
`fuzzer-cmplog-1`	Default	✓	• `-a binary`: handle binary inputs • `-p fast`: favor quick execution paths • `-P explore`: maximize coverage
`fuzzer-cmplog-2`	Default	✓	• `-l1AT`: enable CMPLOG with arithmetic solving and transformational solving • `-P explore`: maximize coverage • `-p explore`: maximize code coverage
`fuzzer-cmplog-3`	Default	✓	• `-Z`: sequential queue selection • `-l1ATX`: CMPLOG with arithmetic, transformation, and extreme solving • `-P explore`: maximize coverage • `-p quad`: quadratic power scheduling • `-P exploit`: prioritize crashes

Table 5.3: Tuning for each one of the 9 fuzzers. Three fuzzers used the Address-Sanitizer (ASan) which uses defensive techniques such as redzones to monitor illegal memory accesses. Similarly, another three fuzzers used Udefined-Behavior Sanitizer (UBSan) in order to detect undefined behavior, as defined in the context of a C program. Finally, in five fuzzers, we enabled the CMPLOG plugin which uses caching and shared memory in order to increase the entropy of the initial seed, thus yield better coverage from early on.

Appendix A.

Source Code Coverage with lcov. As a secondary process which is important for monitoring the effectiveness of our fuzzing methodology, a coverage visualization tool is provided. This tools uses `gcov` [16] and `lcov` [23] in order to create coverage details about the kernel codebase. First, we compile the kernel and syscall invocation method normally, adding the `-coverage` compilation and linking flag. Then we are using all of the stored input corpus generated in the `AFL++` main directory and run the program against it. At the end of this process, we acquire an `lcov` and a html folder with the `lcov` statistics and the web-view of the coverage visualization over the source code. We are also suggest that whether the preferred developer environment is vscode then the user could install the `"Code Coverage LCOV"` plugin [5]. An example of what the web-view main-page looks like is showcased in Figure 5.5. The relative scripts are provided in Appendix A.

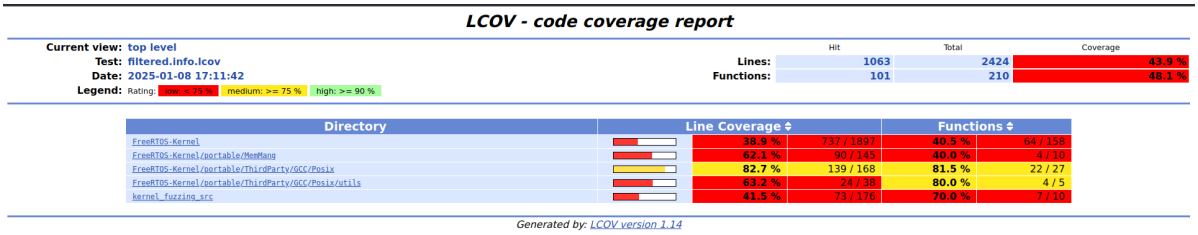

Figure 5.5: `LCOV` generated web-view of coverage after running the whole `AFL++` generated corpus against the syscall routine.

5.3. Evaluation

In this section the effectiveness of the fuzzing methodology is inspected for both the single-thread and multi-threaded approaches. The target of this analysis is to provide the user with some insight about the experimentation process and elaborate on our design choices. Furthermore, the evaluation process can be standardized, enabling a feedback loop that facilitates the better development of a dynamic testing framework.

To ensure a rigorous evaluation of the fuzzing pipeline, **this analysis focuses on the fuzzing campaigns that yielded the maximum number of total crashes** for both single-threaded and multi-threaded harness approaches. By capturing the highest crash rates observed, we establish a comprehensive upper bound on the systems exposure to vulnerabilities. This methodology provides a statistically grounded perspective on the kernel's resilience while minimizing the influence of random variations when evaluating the frameworks efficiency.

FreeRTOS Kernel	Total crashes	Total True Positives	From Valgrind	From gdb	From ASAN
10.6.2	272	265	12	259	173
11.0.0	303	300	9	298	197
11.0.1	295	295	7	291	184
11.1.0	182	180	10	178	135

Table 5.4: Maximum values for total and unique crashes recovered from `AFL++` runs for single-threaded harness. Unique crashes were generated by using unique log output from specified triaging tools. Valgrind, after removing address information, seems to generate the most accurate and precise unique error-log set, without any duplicates and not losing any information.

5.3.1. False Positive and Duplicate Elimination

To eliminate false positives, we re-ran all crashes identified by `AFL++` against our program using GDB, Valgrind, and AddressSanitizer (ASan). The goal of this process was to reduce manual effort by automating crash classification while ensuring that only true, distinct vulnerabilities were considered for further analysis. By extracting meaningful stack traces and memory violation patterns from these tools, we aimed to filter out redundant or misleading crash cases.

For GDB and ASan, we parsed crash reports by analyzing stack traces. However, this process presented

FreeRTOS Kernel	Total crashes	Total True Positives	From Valgrind	From gdb	From ASAN
10.6.2	264	260	12	253	200
11.0.0	238	238	14	236	169
11.0.1	255	255	16	252	189
11.1.0	215	215	15	214	143

Table 5.5: Maximum values for total and unique crashes recovered from `AFL++` runs for multi-threaded harness. Unique crashes were generated by using unique log output from specified triaging tools. Valgrind, after removing address information, seems to generate the most accurate and precise unique error-log set, without any duplicates and not losing any information.

significant challenges due to inconsistencies in reporting formats, particularly with ASan. A major issue encountered was ASans additional memory instrumentation overhead, which, in some cases, caused timeouts in our time-bound execution script, leading to false hang detections. On the other hand, Valgrind proved to be highly effective, as it provided structured call stacks that allowed for precise crash deduplication. By removing memory address-specific information, Valgrind was able to group crashes with identical root causes, significantly reducing the number of unique reports.

Table 5.5 and Table 5.4 summarize the results of this false positive elimination process. The data clearly show that Valgrind-based crash classification outperformed both GDB and ASan in reducing duplicate crash reports, cutting the total count down to a manageable two-digit number. Additionally, ASan demonstrated better performance than GDB in every configuration, likely due to its built-in memory safety checks, which helped in filtering out irrelevant crashes more efficiently.

Notably, while the total number of total and unique crashes varied between multi-threaded and single-threaded fuzzing harnesses, the trend of Valgrinds superior deduplication capabilities remained consistent across both experiments. These results highlight the importance of choosing the right tool for crash triage, as improper classification can lead to significant time wasted on analyzing redundant crash reports.

5.3.2. Coverage Exploration

To assess code coverage, we utilized LCOV with GCCs gcov instrumentation to track executed lines during fuzzing campaigns. Each execution session was configured to capture line-level and function-level coverage, ensuring an accurate representation of how much of the FreeRTOS kernel was exercised by different sanitizers. Coverage was measured using multiple kernel versions, ensuring that differences across releases were accounted for. Categorizing based on versions and sanitizer or cmplog tuning, we were able to gather insights for the effectiveness of our methodology. After the coverage-focused running process was completed, the resulting ".info" files were aggregated and visualized to analyze the percentage of the codebase reached by the fuzzer. This process in combination with the source-code visualization of the coverage provides us with details about the effectiveness of the fuzzing methodology and enables us to better explain the impact of our methodology changes in the overall process.

Table 5.6 presents a comparative analysis of the final coverage percentage obtained for each tested FreeRTOS version across multiple sanitizers. The x-axis represents different kernel versions, while the y-axis denotes the percentage of executed lines. The bars, categorized by different sanitizer configurations, allow us to observe which configurations provide the highest code coverage. Generally, AddressSanitizer (ASan) tends to produce higher coverage due to its ability to detect memory corruption early, allowing fuzzing to continue beyond faults. Lower coverage values in certain configurations indicate early crashes or stagnation in the fuzzing process, preventing deeper exploration of the codebase. Furthermore, the single-threaded and more naive implementation of the harness function yielded lower coverage rates compared to the multi-threaded one as expected. Interestingly, the different fuzzing strategies and the cooperative fuzzing process ensured that all of the fuzzers discover the majority of the paths the rest of the fuzzing executables discovered.

5.3.3. Unique Crashes Exploration

To evaluate unique crashes, we leveraged the aforementioned deduplication scripts. Each crashing input was classified using backtrace analysis and unique fault signatures from gdb, valgrind and ASan-enabled

Final Coverage Reports for FreeRTOS Kernel Versions

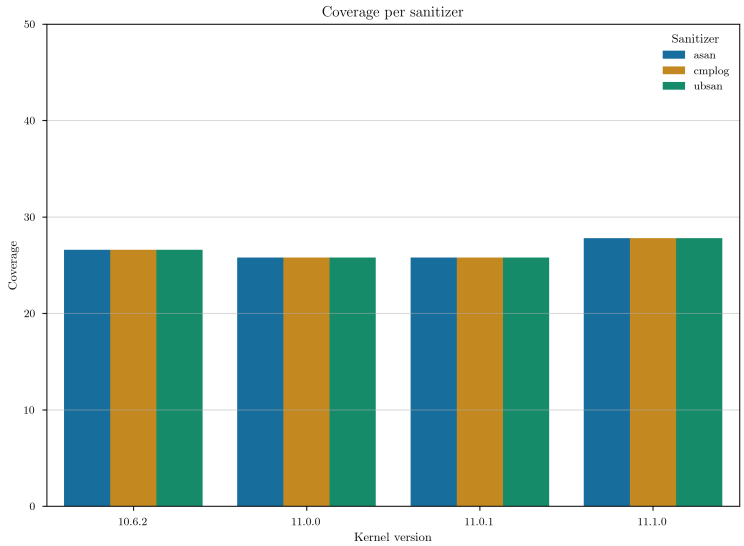

Figure 1: Single-Threaded Coverage Results

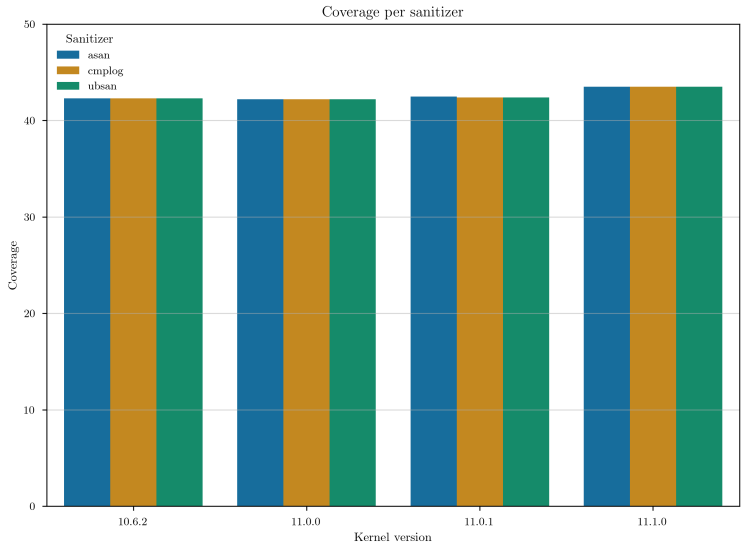

Figure 2: Multi-Threaded Coverage Results

Table 5.6: Coverage reached per distinct fuzzing strategy for different versions of the FreeRTOS Kernel. The results are presented separately for Multi-Threaded and Single-Threaded fuzzing approaches. The results were extracted from the campaign that yielded the higher coverage/crash values.

executions. The goal was to eliminate redundant crashes that stemmed from the same underlying bug while identifying truly distinct vulnerabilities. By filtering crashes based on memory violation types and stack-traces, we ensured that the majority of false positives and duplicates was eliminated. We again categorized based on kernel version and sanitizer that generated the specific crash. Importantly, the effectiveness of different sanitizers does not only lie in the coverage reports but in their ability to discover new crashes.

Final Crash Reports for FreeRTOS Kernel Versions

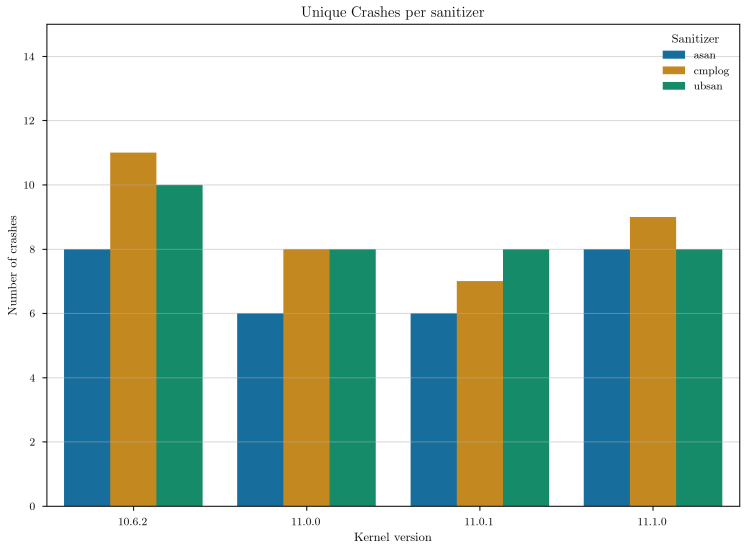

Figure 1: Single-Threaded Crash Reports

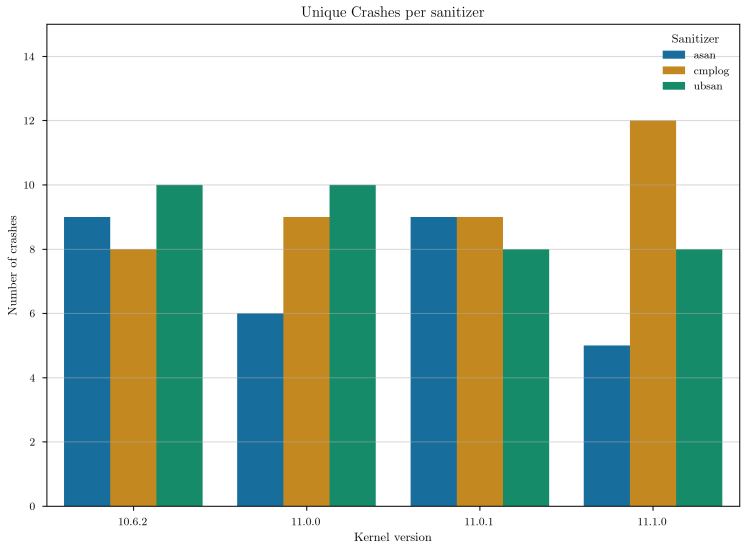

Figure 2: Multi-Threaded Crash Reports

Table 5.7: Unique crashes recorded per different fuzzing strategy for different versions of the FreeRTOS Kernel. The results are presented separately for Multi-Threaded and Single-Threaded fuzzing approaches. The results were extracted from the campaign that yielded the higher coverage/crash values.

Table 5.7 visualizes the number of distinct crash instances across FreeRTOS versions and sanitizer configurations. The x-axis represents different FreeRTOS versions, while the y-axis denotes the number of unique crashes recorded. Each bar is color-coded based on the sanitizer used, highlighting which configurations resulted in the highest number of unique faults. High crash counts in certain sanitizers (e.g., ASan) indicate structural weaknesses in memory safety, while lower values in others suggest stronger protections or fuzzing limitations. Notably, newer FreeRTOS versions tend to have fewer crashes, suggesting that security patches have mitigated some previously discovered issues. According

Crashes Over Executions for FreeRTOS Kernel Versions

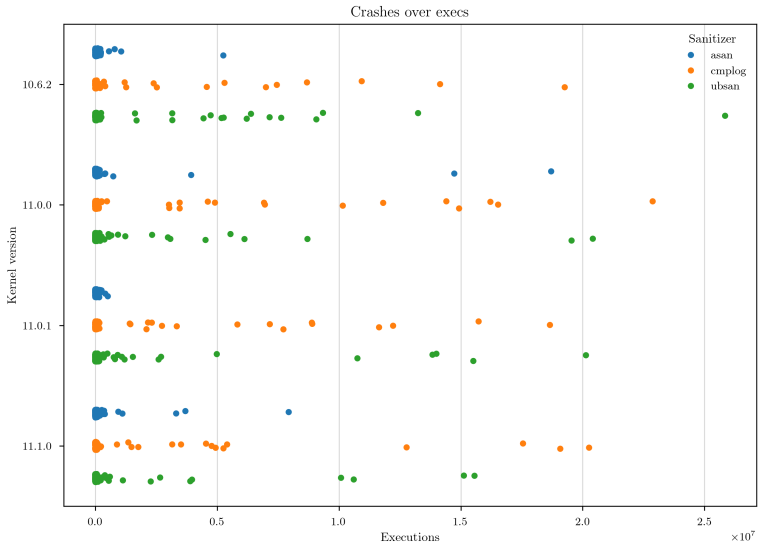

Figure 1: Single-Threaded Crashes Over Executions

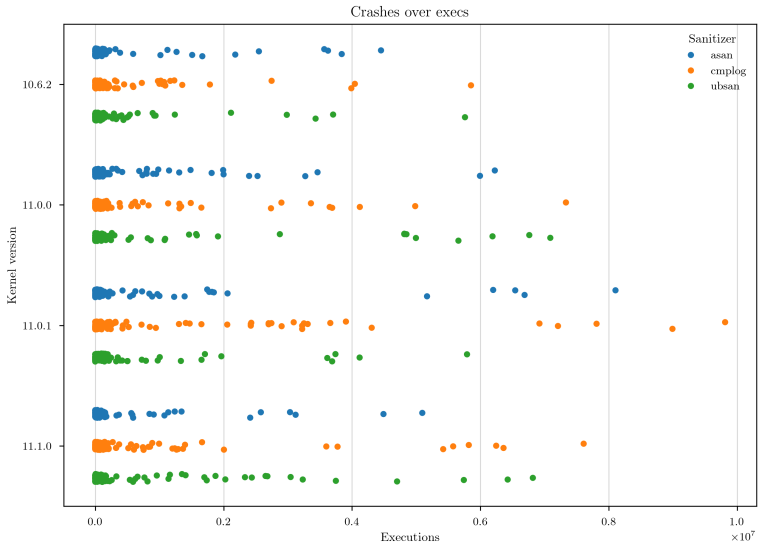

Figure 2: Multi-Threaded Crashes Over Executions

Table 5.8: Crashes recorded over execution cycles for different versions of the FreeRTOS Kernel. The comparison is shown separately for Multi-Threaded and Single-Threaded fuzzing approaches. The results were extracted from the campaign that yielded the higher coverage/crash values. Notably, the multi-threaded harness achieved greater scattering of the crash-over-execution data points, indicating that it scales better than the single-threaded harness. Also, CMPLOG based fuzzing seems to be more consistent in finding new paths, which is supported by the higher coverage values, and therefore is the most scalable of the three distinctive fuzzing groups.

to this metric, CMPLOG enabled execution is the prevalent fuzzing strategy to yield more unique crashes. The sum of unique crashes in Table 5.5 and Table 5.4 is smaller than the aggregation of the unique crashes per sanitizer due to the fact this analysis focuses on the distinct crashes discovered by each sanitizer specifically. Therefore not considering any duplicate crashes discovered by different sanitizers.

5.3.4. Total Crashes over Fuzzer Executions

To evaluate the effectiveness and scaling capabilities of the fuzzing methodology, we tracked crash occurrences over the continuous execution cycles. This metric helps identify whether crashes were occurring consistently or sporadically throughout the fuzzing process. The approach involved exploiting the `AFL++`'s crash-report naming convention which logs number of executions and enables us to correlate them and analyze the rate at which failures emerged. This process also helped us identify whether crashes persisted over time or were resolved as the fuzzer discovered deeper execution paths. It is an important indication to as our harness function continuously discovers new paths and crashes.

Table 5.8 illustrates the distribution of crashes as fuzzing progressed. The x-axis represents the number of executions, while the y-axis denotes cumulative crash count. An expected steep increase in crash count early in the execution suggests shallow, easily triggerable vulnerabilities, whereas a gradual incline indicates deep-state bugs that require extended exploration. If crash discovery plateaus, it may indicate either that all major faults have been found or that the fuzzer is struggling to generate new, effective test cases. This visualization helps determine whether a given configuration is prone to early failures or exhibits progressive fault discovery, both of which are critical factors in assessing the robustness of the FreeRTOS kernel under different conditions. Interestingly, older versions are less prone to deep state bugs, and the fuzzer discovers low hanging vulnerabilities relatively fast, more specifically in the first million executions. Furtermore, we can see that CMPLOG strategy is the one that yields new paths that crash throughout the majority of the execution while ASAN discovers vulerabilities quite fast, since it does not allow memory error to further propagate in the program, therefore failing earlier than the rest of the sanitizers. Nevertheless, in all 4 versions of the kernel and for all three strategies around 85% of the crashing paths is discovered within the first 4 million executions. Again the sparsity of the crashes and the obvious stagnation of the single-threaded harness indicates the lack of complexity and adds to the fact that the more slow but more complex multi-threaded approach yields better results and scales better.

5.4. Vulnerability Reports

Following, we provide vulnerability reports for our findings. Importantly, the last two vulnerabilities, `ISSUE_05` and `ISSUE_06`, where discovered during fuzzing, while the rest were results of software design and static code analysis.

Before getting into details about the vulnerabilities is important to consider the task execution model of the FreeRTOS and try to interpret the task definitions to the Threat Model in Chapter 3. Notably, we discover that the vanilla version of the kernel has certain issues when comes to suitability of the OS to space missions. More importantly our findings showcase the lack of an access control system between tasks and mission-critical system resources, i.e. task priorities, system-calls, etc. Furthermore, the kernel does not define any privilege level for tasks resulting into trusted and untrusted operations to run under the same execution context. At the same time, we consider the MPU version of the kernel which in most of the cases mitigates the issues by enforcing policies over memory regions and a privilege scheme over the task objects. Our suggestions to mitigate relative issues are covered in greated detail in Chapter 6.

5.4.1. ISSUE_00 - `vTaskSuspendAll` Attack

Executive Summary The `vTaskSuspendAll` vulnerability demonstrates the absence of proper scheduler access control in the vanilla FreeRTOS kernel for the POSIX Simulation. An attacker with access to the `vTaskSuspendAll` API call can suspend all other running tasks, leading to significant integrity and availability issues.

Vulnerability Description The lack of access control in the `vTaskSuspendAll` syscall API suspends the scheduler and prevents a context switch from occurring while it allows interrupts during this time. Nevertheless, without manually resuming the scheduler using the `vTaskResumeAll` syscall, the requests are held pending, waiting for the scheduler to resume first. This vulnerability allows an attacker who has access to a task to invoke the `vTaskSuspendAll` API call without any authorization or authentication since the kernel supports no distinction or privilege checks for a task that invokes a scheduler API call. This causes the system's scheduler to suspend all running tasks, with no mechanism to automatically

resume operations, unless the relative syscall is invoked. The lack of access control to the scheduler makes the system susceptible to malicious activity. The affected versions consist of all FreeRTOS Kernel versions displayed in Table 5.1, in which the call is available.

PoC Details The `vTaskSuspendAll` API call halts the execution of all tasks and delays pending interrupts until the scheduler resumes. A malicious task invoking this API call can effectively stop all benign tasks, resulting in complete operational freeze. In our PoC, the benign task was unable to proceed execution beyond the second tick of a ten-second runtime after the malicious task executed the call, as shown by the system-log in Listing 5.1. Additionally, interrupts directed to suspended tasks remain pending, further impacting critical operations.

Listing 5.1: System Log where a malicious task tries to suspend scheduler and a benign task is printing logs periodically.

```
1 benign_task: performing an important operation...
2 benign_task: performing an important operation...
3 benign_task: performing an important operation...
4 benign_task: performing an important operation...
5 malicious_task: suspending all other tasks...
6 malicious_task: all other tasks suspended. If this worked then you should not see any
      more messages after this.
```

Impact Analysis and CVSS Scoring In a mission context, suspending the scheduler causes availability problems. Important periodic tasks are suspended and system interrupts are held pending until the scheduler is resumed, thus vital operations are also interrupted and may cause in abnormalities in the normal mission execution. Besides availability problems, the scheduler's availability and integrity are also compromised and result in failing to keep the hard real-time scheduling promises of the system. This vulnerability requires at least an **Local** attack vector to be deployed in the same execution context as the rest of the tasks. It is a **Low** complexity attack and requires **Low** privileges in order for the vulnerability to be triggered, since the task already runs in the system and all the tasks have the same privilege. It also does not require any interaction from other tasks since it only needs for the malicious task to execute its code to succeed. The scope is not changed. We currently have a PoC for demonstration purposes and have tested the vulnerability in a Simulation Environment, meaning there is **Reasonable** confidence in the report. There is an available mitigation as described in the following paragraph. The final attack vector for this vulnerability is displayed in Table 5.9 and its base score is 6.1.

Possible Mitigation The FreeRTOS MPU kernel extension includes access control that restricts unauthorized tasks from invoking the vulnerable API call [27]. It is recommended to migrate to this secure kernel mode for enhanced protection.

5.4.2. ISSUE_01 - `vTaskPrioritySet` Attack

Executive Summary The `vTaskPrioritySet` vulnerability demonstrates the potential for a malicious task to alter another task's priority, compromising the system's scheduling integrity and availability. By manipulating task priority, the attacker can reduce resource allocation for critical tasks or even render them idle.

Vulnerability Description The lack of access control in `vTaskPrioritySet` syscall sets the priority of any task, given a task handle and a priority number in the range of allowed priorities as defined in the system configuration file. If the kernel is compiled with `INCLUDE_vTaskPrioritySet` set to 1, and a task handle for another task is accessible (via shared variables or leaked information), an attacker in the same execution environment as the target task can alter the latter's priority. This enables the attacker to manipulate system resources allocated to critical tasks by possibly increasing the malicious task priority or even rendering the running tasks idle by setting their priorities to the idle-priority special number (which is 0 in the FreeRTOS case). The affected versions consist of all FreeRTOS Kernel versions displayed in Table 5.1, in which the call is available.

PoC Details A malicious task can invoke the `vTaskPrioritySet` API to change the priority of a benign task. Tasks with priority set to 0 become idle and are excluded from resource scheduling. In our experiment, a benign task performing a critical operation was rendered idle by a malicious task altering its priority as shown by system logs in Listing 5.2. This attack also disrupts the scheduling integrity, significantly impacting operational timelines.

Impact Analysis and CVSS Scoring In a mission context, altering the priorities of other tasks might cause availability problems. Important periodic tasks are missing system resources and their state might change to idle, causing in abnormalities in the normal mission execution. Besides availability problems, the scheduler's integrity is also compromised, which results in possible failure to keep the hard real-time scheduling promises of the original system design. This vulnerability requires at least an **Local** attack vector to be deployed in the same execution context as the rest of the tasks. Due to the fact that a task object handle must be acquired beforehand, which could be done by exploiting another vulnerability or brute force the memory objects in the single address space, this is a **High** complexity attack. It also requires **Low** privilege in order for the vulnerability to be triggered, since all the tasks are running under the same privilege. It also does not require any interaction from other tasks since only the execution of the malicious task is needed for successful exploitation. The scope is not changed. We currently have a PoC for demonstration purposes and have tested the vulnerability in a Simulation Environment, meaning there is **Reasonable** confidence in the report. There is an available mitigation as described in the following paragraph. The final attack vector for this vulnerability is displayed in Table 5.9 and its base score is 5.3.

Possible Mitigation The FreeRTOS MPU kernel extension includes access control that restricts unauthorized tasks from invoking the vulnerable API call [27]. It is recommended to migrate to this secure kernel mode for enhanced protection. When this is not a choice it is suggested to disable the `INCLUDE_vTaskPrioritySet` in the configuration file.

Listing 5.2: System Logs when a malicious task tries to change the priority of the target task. The task handle is leaked through a global variable for simplicity purposes.

```
1 benign_task: priority = 4
2 malicious_task: changing priority of target task
3 malicious_task: changed priority of task 'benign_task' to '2'
4 malicious_task: priority of target task changed. Own priority is '3'
5 benign_task: priority = 2
```

5.4.3. ISSUE_02 - `xTaskAbortDelay` Attack

Executive Summary The discovered vulnerability targets tasks using vTaskDelayUntil, allowing an attacker to disrupt the periodic scheduling of critical operations by invoking a `xTaskAbortDelay` on them. By forcing a delay abort, the attacker compromises task integrity and availability.

Vulnerability Description The `xTaskAbortDelay` API forces a task to leave the blocked state and enter the ready state, even if the task is waiting endlessly on purpose. The syscall is available if the user turns the `INCLUDE_xTaskAbortDelay` option on in the configuration file. The vTaskDelayUntil syscall is used by periodic tasks to ensure constant execution frequency. It is enabled if the `INCLUDE_vTaskDelayUntil` option is turned on in the configuration header file. If an attacker can access the handle of a task blocked on vTaskDelayUntil, they can invoke the `xTaskAbortDelay` API to prematurely terminate the delay period. This disrupts the task's timing consistency, causing operational inconsistencies and compromises the execution frequency. The affected versions consist of all FreeRTOS Kernel versions displayed in Table 5.1, in which the call is available.

PoC Details The `xTaskAbortDelay` API, when invoked, removes the delay block on a target task. In our experiment, a malicious task prematurely aborted the delay of a benign task, resulting in erratic scheduling behavior of the target task without a potential way of restoring the scheduling setting besides a system reset. The target task's periodic operation could no longer maintain a consistent delay frequency, leading to potential system instability.

Impact Analysis and CVSS Scoring This attack compromises both the scheduling integrity of the task and possibly the availability of the system, since critical mission operations, such as data acquisition and vitals monitoring tasks, are usually implemented as periodic tasks. The attack vectors that could trigger the vulnerability are considered **Local**, because we need the attacker process to run in the same execution context as the target task. The attack requires some proper timing in order to invoke the `xTaskAbortDelay` syscall when the target awaits in the blocked state, thus the complexity could be set to **High**. The attack vector requires **Low** privilege in order for the vulnerability to be triggered, since all the tasks are running under the same privilege, and no interaction to succeed. We currently have a PoC for demonstration purposes and have tested the vulnerability in a Simulation Environment, meaning there is **Reasonable** confidence in the report, while there are official mitigation guidelines. The final attack vector for the vulnerability is displayed in Table 5.9 and its base score is 5.3.

Possible Mitigation It is suggested to avoid using vTaskDelayUntil in systems requiring high scheduling integrity, while both `vTaskDelayUntil` and `xTaskAbortDelay` can be disabled by turning off the relative options in the configuration file.

5.4.4. ISSUE_03 - Isolation Issue in Thread Local Storage of FreeRTOS Tasks

Executive Summary An adversary who can use the `vTaskSetThreadLocalStoragePointer` syscall could expose the weak memory isolation in the FreeRTOS kernel. A compromised task can access the thread-local storage of other tasks, compromising memory confidentiality, integrity, and availability.

Vulnerability Description The FreeRTOS kernel configuration enables thread-local storage by defining `configNUM_THREAD_LOCAL_STORAGE_POINTERS`. An attacker who has compromised a task can read and write data in another task's local storage array, bypassing thread local storage access restrictions. In the vanilla kernel version there is no memory isolation between local storage of different tasks. The latter are designed to run as threads and provide an API to read and write values to a system specified memory area. The developer can define the number of different local storage areas in the kernel configuration file. It is discovered that a compromised task can use the `vTaskSetThreadLocalStoragePointer` syscall to set the values of a different task's local storage given that the memory handle is leaked. This is possible due to weak inter-task memory isolation of the thread-like approach to task interpretation by the kernel and the lack of any other access control when it comes to accessing task memory. The affected versions consist of all FreeRTOS Kernel versions displayed in Table 5.1, in which the Thread Local Storage API is implemented.

PoC Details The experiment involved two tasks where the attacker exploited a leaked pointer to access the local storage of a benign task. The attacker was able to read sensitive data and overwrite variables which were crucial to the benign task's execution flow, as displayed in Listing 5.3. For an attacker to perform this attack it should first leak the target address and since FreeRTOS has a single address space model it is easy for accessing another task's private memory because of zero memory-isolation policies applied when the application is running. Importantly this is a data-only attack since the Thread Local Storage is not used to save control-flow related data, but thread-specific sensitive data might contain objects that do change the execution flow of the target task.

Impact Analysis and CVSS Scoring The vulnerability compromises all three aspects of the local storage CIA, since the attacker can read and write data but in may also cause availability issues. The attack vectors should be **Local**, since we need the two tasks, attacker and target, to run in the same execution context and it is considered a **High** complexity attack which needs **Low** privilege in order for the vulnerability to be triggered, since all the tasks are running under the same privilege and the threat. The attack does not require any kind of interaction to execute and the scope remains unchanged. We currently have a PoC demonstrating the attack in a Simulation Environment, meaning there is **Reasonable** confidence in the report. Finally, there are official mitigation guidelines to mitigate the vulnerability. The derived attack vector is displayed in Table 5.9 and its base score is 7.0.

Possible Mitigation It is suggested to migrate the kernel to the MPU version which feature a memory management unit and allows to create process-wide memory access policies [27]. Again this does not

solve the issue for tasks under the same memory-access policy. As a further suggestion, sensitive data should not be stored in the task local storage, or if it necessary, an data integrity mechanism should be employed.

Listing 5.3: Output demonstrating unauthorized access and modification of another task's local storage

```
benign_task: saving secret value in local storage...
benign_task: secret value is deadbeef
malicious_task: leaked secret value deadbeef
malicious_task: Changing secret of another thread to 0x80085
benign_task: secret value is 80085
```

5.4.5. ISSUE_04 - Static Task Stack Compromise

Executive Summary The static task stack compromise demonstrates that FreeRTOS lacks robust access controls for task stacks. An attacker can access and modify another task's stack variables, compromising memory CIA and possibly the target's control flow.

Vulnerability Description The FreeRTOS Kernel provides static task resource allocation and creation through the `xTaskCreateStatic` syscall. The user can define the task's stack and supply it with the only restriction that this memory space is defined as a static variable, meaning it is persistent among calls. This stack is operating as a normall stack allowing the task to store local variables and control flow metadata to facilitate its execution. A malicious task can exploit a leaked pointer to a benign task's stack to read or modify its variables. This can lead to the leakage of sensitive data or the alteration of control-flow variables, allowing unauthorized behavior. The affected versions consist of all the versions of FreeRTOS kernel displayed in Table 5.1.

PoC Details During the experiment, an attacker used a leaked stack variable pointer to overwrite the benign task's local variables. The leaked object handle could be acquired by exploiting a memory leak vulnerability or by bruteforcing static objects in the single address space. This not only compromised sensitive data but also altered execution flow, enabling unauthorized task operations. Further exploitation could completely compromise the CIA of the target task stack, leading to possible Control-Flow Integrity issues. The single address space memory model of FreeRTOS dictates zero memory isolation boundaries on inter-task memory accesses, thus enabling this attack.

Impact Analysis and CVSS Scoring The vulnerability compromises all three aspects of CIA for the task static memory. The attack vector that triggers this vulnerability is **Local**, since we need the two tasks, attacker and target, to run in the same execution context and memory address space and it is considered a **High Complexity** attack, since it requires to leak the stack's handle in order to exploit the vulnerability. The attack vector needs **Low** privilege to execute and does not require any kind of interaction to execute, while the scope remains unchanged. We currently have a PoC demonstrating the attack in a Simulation Environment, meaning there is **Reasonable** confidence in the report. Finally, there are official mitigation guidelines to mitigate the vulnerability. The derived attack vector is displayed in Table 5.9 and its base score is 7.0.

Possible Mitigation The FreeRTOS MPU version supports privilege-based memory isolation between tasks through its memory management unit and is therefore the officially suggested mitigation due to its capability of creating process-wide memory access policies that could mitigate this issue if implemented properly [27]. Again this does not solve the tasks for tasks under the same memory-access policy. It is also considered a good practice to never store sensitive data in the stack without some integrity checking mechanism employed.

5.4.6. ISSUE_05 - Data Overwrite via xTaskCreateStatic

Executive Summary Through our manual exploration of the crash-logs from fuzzing the FreeRTOS kernel, we discovered that the `xTaskCreateStatic` API is vulnerable to buffer overflow data-only attacks, allowing attackers to overwrite adjacent memory regions and potentially sensitive variables of their choice. This compromises the kernel-user-space separation and the memory integrity of sensitive data.

Vulnerability Description The `xTaskCreateStatic` API allows attackers to specify a stack buffer and, most importantly, size during task creation. By providing an size value larger than the actual size of the provided buffer, attackers can write beyond the buffer boundary, potentially altering control variables or sensitive data. More specifically, the API's implementation uses a `memset` call to initialize the buffer for which the size parameter is the same as the user defined one. If the size is larger then the initialization byte is written beyond the bounds of the buffer, overwriting adjacent memory locations. The affected versions consist of all the FreeRTOS Kernel versions mentioned in Table 5.1.

PoC Details In the experiment, an attacker used the `xTaskCreateStatic` API to overwrite adjacent memory regions with the byte value `0xA5`, which is the byte that `memset` uses to initialize the static stack parameter. For the attack to be successful the target variable should be adjacent to the attacker controlled variable and stored after the latter in the relative memory layout. In our example, we used the problematic syscall to alter the value of a global target variable when the attacker controlled an adjacent global variable, both of which were saved in the uninitialized segment (`.bss`) of the program.

Impact Analysis and CVSS Scoring Although the written value is not attacker-controlled, this overflow has significant security implications to the memory integrity and possibly to the execution's integrity if the variable is used to control the operations of the running task. Nevertheless the attacker cannot control the data that are overwritten, therefore its impact is deemed as Low. Again the attack vector is **Local**, since the compromised task must be run in the same context with the target task and perform the ill-formed syscall to overwrite the target memory object. The complexity is therefore **High** and the attacker process could be of **Low** privilege, while no interaction is needed from the target task to trigger the vulnerability and the scope does not change. We currently have a PoC program that demonstrates an attack scenario in a Simulation Environment, meaning there is **Reasonable** confidence in the report, while there is an official partial mitigation for this vulnerability. The final attack vector is displayed in Table 5.9 and its base score is 3.6.

Possible Mitigation The FreeRTOS MPU version supports privilege-based memory isolation between tasks through its memory management unit and is therefore the officially suggested mitigation when high privileged task isolation is required. This can be achieved through its MPU capability that allows the creation of process-wide memory access policies [27]. Again this does not solve the problem for tasks that run under the same memory policy. Another possible mitigation, whether the user decides to use the vanilla kernel version, is to restrict the use of this call to the unprivileged tasks that are not supposed to create tasks.

5.4.7. ISSUE_06 - Data Overwrite via xStreamBufferCreateStatic

Executive Summary Through our manual exploration of the crash-logs from fuzzing the FreeRTOS kernel, we discovered that the `xStreamBufferCreateStatic` API is vulnerable to buffer overflow data-only attacks, allowing attackers to overwrite adjacent memory regions and potentially sensitive variables of their choice. This compromises the kernel-userspace separation and the memory integrity of sensitive data.

Vulnerability Description The `xStreamBufferCreateStatic` API is used to create static stream buffer data structures for Inter-Process Communication (IPC). It allows users to specify a buffer that acts like a buffering stream, from which tasks can read from and write to, and the size of the buffer during stream-buffer creation. By providing an size value larger than the actual size of the provided buffer, attackers can write beyond the buffer boundary, potentially altering sensitive data. More specifically, the API's implementation uses a `memset` call to initialize the buffer for which the size parameter is the same as the user defined one. If the size is larger then the initialization byte is written beyond the bounds of the buffer, overwriting adjacent memory locations. The affected versions consist of all the FreeRTOS Kernel versions mentioned in Table 5.1.

PoC Details In the experiment, an attacker used the `xStreamBufferCreateStatic` API to overwrite adjacent memory regions with the byte value `0x55`, which is the byte that `memset` uses to initialize the static stream buffer object, provided as a parameter to the call. For the attack to be successful the

Vulnerability	Vector	Base Score	Temporal Score	Environmental Score
ISSUE_00	CVSS:3.0/AV:L/AC:L/PR:L/UI:N/ S:U/C:N/I:L/A:H/E:P/RL:O/RC:R/ CR:H/IR:H/AR:H/MAC:H/MPR:H	6.1	5.3	5.4
ISSUE_01	CVSS:3.0/AV:L/AC:H/PR:L/UI:N/ S:U/C:N/I:L/A:H/E:P/RL:O/RC:R/ CR:H/IR:H/AR:H/MAC:H/MPR:H	5.3	4.6	5.4
ISSUE_02	CVSS:3.0/AV:L/AC:H/PR:L/UI:N/ S:U/C:N/I:L/A:H/E:P/RL:T/RC:R/ CR:H/IR:H/AR:H/MAC:H/MPR:H	5.3	4.6	5.4
ISSUE_03	CVSS:3.0/AV:L/AC:H/PR:L/UI:N/ S:U/C:H/I:H/A:H/E:P/RL:O/RC:R/ CR:H/IR:H/AR:H/MAC:H/MPR:H	7.0	6.1	5.5
ISSUE_04	CVSS:3.0/AV:L/AC:H/PR:L/UI:N/ S:U/C:H/I:H/A:H/E:P/RL:O/RC:R/ CR:H/IR:H/AR:H/MAC:H/MPR:H	7.0	6.1	5.5
ISSUE_05	CVSS:3.0/AV:L/AC:H/PR:L/UI:N/ S:U/C:N/I:L/A:L/E:P/RL:O/RC:R/ CR:H/IR:H/AR:H/MAC:H/MPR:H	3.6	3.1	3.5
ISSUE_06	CVSS:3.0/AV:L/AC:H/PR:L/UI:N/ S:U/C:N/I:L/A:L/E:P/RL:O/RC:R/ CR:H/IR:H/AR:H/MAC:H/MPR:H	3.6	3.1	3.5

Table 5.9: CVSS Vectors for FreeRTOS findings. For the environmental Scores we set the CIA Requirements to High for all of the vulnerabilities, while for ISSUE_01 and ISSUE_02, we set the Modified Privileges required to High to indicate that the configurations needed to make the vulnerable methods available are not the default ones supported by the FreeRTOS Kernel.

target variable should be adjacent to the attacker controlled variable and stored after the latter in the relative memory layout. In our example, we used the problematic syscall to alter the value of a global target variable when the attacker controlled an adjacent global variable, both of which were saved in the uninitialized segment (`.bss`) of the program.

Impact Analysis and CVSS Scoring Although the written value is not attacker-controlled, this overflow has significant security implications to the memory integrity and possibly to the execution's integrity if the variable is used to control the operations of the running task. Nevertheless the attacker cannot control the data that are overwritten, therefore its impact is deemed as Low. Again the attack vector is **Local**, since the compromised task must be run in the same context with the target task and perform the ill-formed syscall to overwrite the target memory object. The complexity is therefore **High** and the attacker process could be of **Low** privilege, while no interaction is needed from the target task to trigger the vulnerability and the scope does not change. We currently have a PoC program that demonstrates an attack scenario in a Simulation Environment, meaning there is **Reasonable** confidence in the report, while there is an official partial mitigation for this vulnerability. The final attack vector is displayed in Table 5.9 and its base score is 3.6.

Possible Mitigation The FreeRTOS MPU version supports privilege-based memory isolation between tasks through its memory management unit and is therefore the officially suggested mitigation when high privileged task isolation is required. This can be achieved through its MPU capability that allows the creation of process-wide memory access policies [27]. Again this does not solve the problem for tasks that run under the same memory policy. Another possible mitigation is to restrict the use of this call to the unprivileged tasks that are not supposed to create IPC objects.

5.4.8. Setting the Environmental Scores

In Table 5.9, we define the environmental scores for the identified vulnerabilities, consistently setting both "Modified Attack Complexity" (MAC) and "Modified Privilege Required" (MPR) to "High." The rationale behind this approach stems from the highly specific execution environment of space systems. Both hardware and software configurations are fine-tuned to meet mission-specific requirements, making it significantly more challenging for an attacker to understand the exact conditions necessary for a successful exploit. For instance, in `ISSUE_05` and `ISSUE_06`, the vulnerabilities depend on the way variables are stored in memory. Proof-of-concept (PoC) exploits that work in a simulated environment may fail on an actual satellite platform due to incorrect assumptions about how memory is managed in the deployed system.

Additionally, our threat model assumes that the attacker already has control over a compromised task. In the vanilla FreeRTOS kernel, all tasks share the same privilege level due to its thread-based design, meaning that an accurate privilege assessment must consider the actual level of access required to execute the attack and its real impact. Unlike traditional OSes, FreeRTOS does not distinguish between user-space and kernel-space execution, effectively granting every task "High" privilege by default. Furthermore, since tasks inherit the same privilege level assigned to the system by the hypervisor, they have unrestricted access to system resources. As a result, we set "Modified Privilege Required" to "High" to reflect this unrestricted execution model.

By adjusting these environmental score metrics, we aim to accurately represent the likelihood and feasibility of an attack, ensuring that the overall CVSS score remains appropriately balanced. Our research specifically focuses on post-exploitation and onboard exploitation scenarios, which inherently occur later in the attack chain and demand a deeper understanding of the target platform and software to be carried out successfully.

5.5. Cube-FlatSAT Testing

In order to evaluate the aforementioned issues, we compile and run an image of the PoCs on a Cube-FlatSAT created for testing and experimentation. The process consists of running the PoCs to an OBC simulator and then cross compiling the images and flashing them into the flatsat OBC. For retrieving result evidence we create a logging server that gathers logs sent over a CAN-Bus which is connected to the ground section. Afterwards, we determine how many of the PoCs actually work on the platform and re-evaluate the CVSS scoring by changing the "Exploit Code Maturity" scoring.

The Cube-FlatSAT runs FreeRTOS 10.2.6, which is older than the tested versions of the software but contains the dependencies needed to run the majority of our PoCs. We utilize the OBC source code created for the 2024 3S Security Challenge [45] which uses CubeSat Space Protocol (CSP) [24] for communication between the 4 computing modules using a CAN-Bus. This CAN-Bus is also available to the Ground Segment through a CAN-to-USB adaptor which enables us to tap into the interface and simulate a CSP node which participate in the protocol.

Firstly, we integrate our benign and malicious tasks in the simulator, compile the OBC and run it using the POSIX Simulation for FreeRTOS. Once this step is completed, we adjust the OBC simulator to send logs over the CAN-Bus to a special node and port numbers, the ones used to deploy our logging server. This way every time a progress message is communicated the Ground Segment takes note and enables us to retrieve evidence of what is happening on-board. Afterwards, we compile the OBC source-code with the arm cross-compiler and use C-Shell to communicate with the on-board nodes and flash our software in one of them. With the logging server listening for connections, we store logs of the messages receive and inspect them later in order to evaluate the success of our experiments.

The logging server consists of a C script that sets up the CSP interface, initializes the CSP protocol and routing services and then listens indefinitely to new connections. If the connection is targeting the agreed-upon port, the server logs the message, otherwise it default to the predefined behavior so that it does not lose any default functionality. The logs are stored locally on the running machine. The service should be run with administrator privileges so that it can re-initialize the CAN interface before starting the server.

From the starting POCs only `ISSUE_05` and `ISSUE_06` failed to successfully execute on the representative

Vulnerability	Vector	Base Score	Temporal Score	Environmental Score
ISSUE_00	CVSS:3.0/AV:L/AC:L/PR:L/UI:N/ S:U/C:N/I:L/A:H/E:H/RL:O/RC:R/ CR:H/IR:H/AR:H/MAC:H/MPR:H	6.1	5.6	5.7
ISSUE_01	CVSS:3.0/AV:L/AC:H/PR:L/UI:N/ S:U/C:N/I:L/A:H/E:H/RL:O/RC:R/ CR:H/IR:H/AR:H/MAC:H/MPR:H	5.3	4.9	5.7
ISSUE_02	CVSS:3.0/AV:L/AC:H/PR:L/UI:N/ S:U/C:N/I:L/A:H/E:H/RL:T/RC:R/ CR:H/IR:H/AR:H/MAC:H/MPR:H	5.3	4.9	5.7
ISSUE_03	CVSS:3.0/AV:L/AC:H/PR:L/UI:N/ S:U/C:H/I:H/A:H/E:H/RL:O/RC:R/ CR:H/IR:H/AR:H/MAC:H/MPR:H	7.0	6.4	5.9
ISSUE_04	CVSS:3.0/AV:L/AC:H/PR:L/UI:N/ S:U/C:H/I:H/A:H/E:H/RL:O/RC:R/ CR:H/IR:H/AR:H/MAC:H/MPR:H	7.0	6.4	5.9

Table 5.10: Updated CVSS Vectors for FreeRTOS findings. The last two findings were dismissed since there is no change. The "Exploit Code Maturity" metric is properly adjusted for the rest.

hardware. For the first one, the reasons are architecture specific, since it dependes on the order of the affected variables-assets in the different data sections (`.bss`, stack, etc.), while the second one was not able to compile in the first place since the version was not supporting the syscall-api we were trying to exploit. In Table 5.10 we see the new CVSS scores for the issues which were effectively demonstrated on the Cube-FlatSAT. The "Exploit Code Maturity" scores change from "Proof-Of-Concept" to "High" since the PoCs are also working in the specialized testing platform, while the "Report Confidence" remains "Reasonable" because we did not test against a real OBSW of a space mission.

More interestingly `ISSUE_00` caused the Watchdog task to suspend its execution. The user-level component "SW Watchdog", as displayed in Figure 5.3, consists of a task that handles communication with the supervisor technology of the flatsat, making sure that it responds to commands and/or health-pings from the former. By suspending this task, the health ping fails and we trick the supervisor to perform a soft reboot on the node while it also changes the selected FLASH image that is loaded to run on the node. In our case, if we flash this image to all 4 available flashes we cause the satellite to enter a state that it will always reboot and try to find a "healthy" image to boot. More specifically, if we want to force the node to run a specific image that we have loaded in a fixed FLASH module, we could use this issue to block the rest of the images from running.

5.6. Limitations

Fuzzing Fuzzing the FreeRTOS Kernel using the POSIX Simulator resulted in constructing a series of tools and scripts to automate the process and enhance the static analysis and code auditing phases. Nevertheless, our fuzzing methodology was based on `AFL++` a generic method coverage guided framework which does not facilitate state and context awareness and does not support grammar-based fuzzing of any kind out of the box. Therefore, we had to tune the framework and create middle-ware that enabled us to simulate different tasks running concurrently. The adjustments made, although they enhanced the achieved coverage and comprehensiveness of our testing, they slowed down the fuzzing significantly. Furthermore, fuzzing the system-level state is a challenging and complex task, especially when using a generic use fuzzer such as `AFL++`. Although out of the scope of the current work, focusing in fuzzing system state would improve the efficiency of our fuzzing and enable us to explore different kinds of bugs, e.g. race conditions. Finally, both the syscall and parameter selection for fuzzing is based on the simplicity of the fuzzed parameters, an action which, on its own, discarded a fair amount of API calls from our checks because of the complexity of meaningfully fuzz parameters such as memory objects and pointers, which resulted in generating a large amount of early-crashing inputs, thus were fixed in order

to enable deeper exploration of the instrumentation trees for the selected syscalls. The integration of state-of-art real-time-system-state supporting fuzzers [38] could tackle this challenge and enable our harness function to fuzz more sophisticated syscalls on which object handles and their internal state are an important factor which guides the syscall execution path.

Cube-FlatSAT Testing The Cube-FlatSAT setup introduced several limitations that affected the testing and validation process. One major challenge was the lack of a robust debugging toolset, making it difficult to retrieve meaningful results when FreeRTOS-based experiments caused system crashes or unexpected behavior. Additionally, the FreeRTOS version running on the platform was outdated (10.2.6), which limited compatibility with some of the proof-of-concept (PoC) exploits. Specifically, one PoC could not be tested due to missing API support in this older version. Another significant issue was the impact of soft reboots on debugging consistency each time the OBC crashed, it flushed to a different image, making it difficult to maintain a persistent state for analysis. This constant switching between images disrupted the ability to track failures systematically and slowed down the iterative testing process. These factors collectively hindered a more comprehensive evaluation of the vulnerabilities in a real-world onboard system.

6 Discussion

6.1. Responsible Discloure

Responsible disclosure is a key practice in vulnerability research, ensuring that security flaws are communicated to relevant stakeholders in a structured and ethical manner before they can be exploited maliciously. The common process for disclosing vulnerabilities involves privately reporting findings to the affected vendor or governing body, allowing them adequate time to investigate, patch, and mitigate risks before public disclosure. This is typically done through a Coordinated Vulnerability Disclosure (CVD) process, where researchers provide a detailed technical report, including proof-of-concept exploits, impact assessments, and potential mitigations. Depending on the severity and scope of the issue, the organization receiving the report may work with national cybersecurity agencies, standardization bodies, or sector-specific security groups to address the vulnerabilities comprehensively.

Given the mission-specific nature of vulnerabilities in space systems, where software security risks are tightly coupled with onboard hardware configurations, operational constraints, and mission objectives, a traditional vulnerability disclosure approach is not always feasible.

To ensure our research findings were responsibly handled, we engaged in direct communication with the European Space Agency (ESA) and submitted our vulnerability reports to their security teams. ESA, as a key actor in European space security, provided valuable feedback on our findings, particularly in evaluating CVSS scoring and setting Environmental Scores to better reflect real-world risk in the context of space missions. This collaborative process helped refine the severity ratings of our findings by incorporating domain-specific considerations such as mission duration, onboard software update feasibility, and isolation guarantees provided by spacecraft architectures.

By following this disclosure process, we aimed to contribute constructively to the security of spaceborne Real-Time Operating Systems and hypervisors, ensuring that potential threats are mitigated before they can be leveraged in an operational environment.

6.2. SPACE-SHIELD Contributions

In this work we also aim to extend the existing SPACE-SHIELD framework [41] with new techniques and impact scenarios that would also reflect to our experiments. Our contribution is partially inspired by the SPARTA [43] and the MITRE EMB3D [28] frameworks.

Lateral Movement

We propose two new sub-techniques under the "Compromise the satellite platform starting from a compromised payload" category:

- **Inter-Task Compromise**: This attack path involves compromising a single task within an RTOS and leveraging inadequate memory isolation to influence or control other tasks. In our experiments, this was demonstrated through the attack that exploited the lack of access control in

`vTaskPrioritySet` system call, enabling a compromised task could manipulate the scheduling and execution of others.

- **Inter-Application Compromise**: In hypervisor environments, compromising one guest application could lead to unauthorized interactions with other applications, exploiting weaknesses in spatial or temporal isolation. This was evidenced by attacks exploiting the hypervisor's scheduling policies.

Impact

We introduce two new impact scenarios under the "Saturation/Exhaustion of Spacecraft Resources" technique:

- **RTOS Scheduler Compromise**: This involves manipulating the RTOS scheduler to delay or deny the execution of critical tasks, potentially impacting satellite operations like Attitude Determination and Control System (ADCS) or Electrical Power System (EPS) management.
- **Hypervisor Scheduling Compromise**: Similar to the RTOS scenario but within hypervisor environments, where malicious manipulation of partition schedules can disrupt mixed-criticality operations.

Execution

A new technique, "Exploit Code Flaws", is introduced, with further refinements tailored to Space OS and is inspired by the similar category described in the SPARTA framework. The following sub-techniques are the first suggested candidates that could fit this classification of attack scenarios:

- **(RT)OS Exploits**, which focus on compromise of assets of the (Real Time) Operating System running on the OBC modules. They could range from memory corruption exploits up to access control abuse. Based on our experiments and use case we can also make more concrete techniques such as **Lack of Access Control in Task Memory** and **Overwriting Memory Areas through Buffer Overflows**.
- **Hypervisor Software Exploits**, which focus on exploiting bugs in the hypervisor software and gain access to this process. Although escaping techniques are considered part of lateral movement, there is a change of domain in this case, thus we include this group of techniques in the arbitrary execution class.
- **Known Vulnerability Exploits**: Systematic exploitation of publicly disclosed CVEs relevant to Space OS.

These extensions enhance the SPACE-SHIELD frameworks capability to map complex attack vectors specific to RTOS and hypervisor systems, reflecting realistic threats in space missions. Furthermore, by integrating these attack paths, SPACE-SHIELD aligns more closely with the detailed classifications found in SPARTA.

Table 6.1 demonstrates the extensions needed in the SPACE-SHIELD framework to effectively cover the vulnerabilities identified in our research. Notably, many of the RTOS-related attacks could not explicitly be mapped in the current version of SPACE-SHIELD and SPARTA frameworks. Our proposed extensions, such as the "RTOS Scheduler Compromise" and "Exploit Code Flaws" categories, provide a more granular classification for these vulnerabilities. These additions are essential for accurately modeling threat vectors in Space OS environments, enhancing the comprehensiveness of SPACE-SHIELD and aligning it more closely with real-world attack scenarios.

6.3. Mitigation Suggestions

To effectively secure Space OS platforms, particularly RTOS and Hypervisors in mixed-criticality satellite systems, a comprehensive set of mitigation strategies is required. This section presents general design guidelines as well as specific countermeasures aimed at neutralizing identified attack vectors. These recommendations are grounded in both industry standards and recent research findings, including sandboxing, access control mechanisms, and secure communication protocols.

Ensuring strict memory isolation between tasks and partitions is essential to prevent unauthorized access and mitigate data leakage. In RTOS environments, this can be achieved using Memory Protection Units (MPUs) to segregate task memory spaces. For hypervisors, spatial isolation mechanisms must

Vulnerability ID	SPACE-SHIELD Mapping	SPARTA Mapping	Adjusted SPACE-SHIELD Mapping
ISSUE_00	N/A	Modify On Board Values: Scheduling Algorithm	Impact: RTOS Scheduler Compromise
ISSUE_01	N/A	N/A	Lateral Movement: Inter-Task Compromise
ISSUE_02	N/A	Modify On Board Values: Scheduling Algorithm	Impact: RTOS Scheduler Compromise
ISSUE_03	Software Vulnerabilities	Exploit Code Flaws: Operating System	Execution: (RT)OS Exploits: Lack of Access Control in Task Memory
ISSUE_04	Software Vulnerabilities	Exploit Code Flaws: Operating System	Execution: (RT)OS Exploits: Lack of Access Control in Task Memory
ISSUE_05	Software Vulnerabilities	Exploit Code Flaws: Operating System	Execution: (RT)OS Exploits: Overwriting Memory Areas through Buffer Overflows
ISSUE_06	Software Vulnerabilities	Exploit Code Flaws: Operating System	Execution: (RT)OS Exploits: Overwriting Memory Areas through Buffer Overflows

Table 6.1: Mapping of Vulnerabilities to SPACE-SHIELD and SPARTA Frameworks, including Proposed Extensions

be enforced, particularly between partitions of different criticality levels, to safeguard sensitive data and prevent lateral movement attacks. Moreover, temporal isolation through time-partitioned scheduling models can ensure high-priority tasks receive the necessary CPU time without interference from compromised low-priority tasks.

Sandboxing To mitigate risks from compromised applications, especially in multi-tenant platforms, sandboxing and access control mechanisms can be employed to restrict privileges and system access at both the application and system levels. These approaches isolate malicious code within a controlled environment, preventing lateral movement, unauthorized system modifications, and privilege escalation. In Linux-based environments, tools such as seccomp-bpf, cgroups, and namespaces effectively limit process privileges and access to system resources, confining execution within restricted environments and preventing unauthorized interactions with critical system components [26]. For CubeSat missions, seccomp-bpf filters are particularly useful for restricting dangerous system calls, while cgroups enforce resource constraints, mitigating the risk of denial-of-service (DoS) attacks from malicious processes. To further enforce the principle of least privilege, Mandatory Access Control (MAC) systems such as AppArmor and SELinux regulate file and resource access permissions, minimizing the attack surface. These systems enforce capability-based privilege separation, restricting the actions each application can perform. Their effectiveness is enhanced when combined with containerization tools like nsjail and firejail, which provide additional namespace segregation and syscall filtering.

Memory Protection and Error Correction Memory corruption, including Single-Event Upsets (SEUs) induced by cosmic radiation, poses a critical threat to satellite systems. To mitigate this, a combination of hardware and software countermeasures is necessary. Utilizing Error Correction Codes (ECC) provides error detection and automatic correction of single-bit errors, maintaining data integrity. Additionally, deploying radiation-hardened processors and memory modules enhances resilience against cosmic radiation-induced faults, ensuring system stability and reliability [26, 54]. Furthermore, implementing protected memory regions in RTOS and hypervisors prevents attackers from accessing or modifying critical data structures, safeguarding system integrity and confidentiality.

Privilege Escalation Mitigation To safeguard against privilege escalation and resource abuse, control mechanisms are necessary to limit process resource consumption and access levels. In Linux-based environments, cgroups can enforce resource usage limits, preventing a compromised process from monopolizing CPU, memory, or I/O resources. This mitigates the risk of denial-of-service (DoS) attacks by containing resource abuse within isolated groups [26].

Secure-by-Design Adopting a Secure-by-Component design paradigm ensures that each system module is isolated and protected, minimizing the risk of cascading failures or privilege escalation attacks. By designing space systems as a collection of isolated subsystems, such as command and control, payload operations, and communication modules, the blast radius of a potential attack is minimized [50, 54]. Implementing layered security controls, including access control policies, encryption, and hardware-level isolation, ensures that an attack on one component cannot propagate to other critical modules.

FreeRTOS MPU Case Since the focus of the experiments was on FreeRTOS Kernel, we explored in detail the capabilities of the MPU-enhanced kernel [27] as a potential mitigation for the issues we identified in our experiments. The MPU-enhanced FreeRTOS kernel enforces memory access policies between tasks by restricting direct memory access outside predefined regions. This mechanism mitigates memory corruption risks by isolating task stacks, preventing buffer overflows from propagating between tasks. It managed to mitigate all of the discovered issues in a simulated environment. Furthermore, it introduces a distinction between privileged and unprivileged tasks, allowing fine-grained control over system resources. Additionally, "restricted" tasks enforce permissions over shared memory regions, limiting their ability to interfere with critical system components. Restricted tasks are also prohibited from executing dangerous system calls, ensuring that a compromised low-privilege task cannot manipulate the scheduler or escalate its privileges. This effectively mitigates some of the vulnerabilities we reported in the vanilla kernel, such as scheduler manipulation through `vTaskSuspendAll` or unauthorized priority modifications via `vTaskPrioritySet`. However, it is important to note that MPU enforcement is hardware-dependent and requires a processor with an MPU unit, which might not be available on all space-qualified microcontrollers. Additionally, MPU-based enforcement applies only to tasks and does not extend to kernel-level components, meaning that a vulnerability in a privileged task could still compromise the system. Moreover, actors such as Interrupt Service Routines (ISRs) and peripheral device drivers remain fully trusted and are not within the scope of the MPU security model, potentially leaving them as an attack surface for privileged escalation through driver exploits.

7

Related Work

Security assessment of on-board software, and specifically Real-Time Operating Systems (RTOS), has gained increasing attention in recent years due to the critical role these components play in different kinds of embedded applications, one of which would be space missions. While extensive research has been conducted on security testing methodologies for general-purpose operating systems in terrestrial environments, significantly less work has been done to address the unique constraints of space systems [13, 18, 53, 54]. This chapter explores the state-of-the-art work in RTOS security testing, highlighting the methodologies employed, key vulnerabilities discovered, and the gaps that remain in the field. Furthermore, we examine open-source vulnerability research, which provides valuable insights into existing security weaknesses in widely used space software, emphasizing the urgent need for a standardized security testing framework.

7.1. RTOS Testing

Similarly to hypervisors, dynamic analysis of Real-Time Operating Systems is a rather interesting and again challenging field. RTOS are more common when hard time constrains are required, therefore availability and scheduling integrity are important assets and usual targets for attackers. Tools such as SFuzz and RTKaller perform dynamic analysis utilizing advanced fuzzing techniques that are adjusted to the multi-tasking environment of the RTOS landscape.

SFuzz, a slice-based fuzzing framework, has demonstrated significant success in identifying vulnerabilities in RTOS binaries by isolating functionality-specific code slices. Using forward and backward slicing, SFuzz achieves precise targeting of potential weaknesses, identifying 77 previously unknown vulnerabilities, with 67 assigned CVEs, showcasing arbitrary code execution and Denial of Service attack vectors on the tested RTOS [4]. Similarly, Rtkaller employs a state-aware fuzzing methodology to test real-time-specific code, which traditional kernel fuzzers often fail to explore. By generating task-based test cases, Rtkaller uncovered 28 new vulnerabilities including memory corruption and race condition attack vectors, emphasizing the importance of state-sensitive testing [38].

Both tools start by highlighting the challenges which are inherent in RTOS testing due to their monolithic architecture and lack of strict separation between kernel and user space. Futhermore, they try to explore alterations on the fuzzing methodology, instrumentation and symbolic execution phases to increase the general performance. These design characteristics necessitate dynamic and context-sensitive testing strategies that account for inter-task dependencies and real-time constraints. Nevertheless, in our research we prioritize the experimentation with standard industry tools and assume that advanced and non-industry tested solutions are out of scope. The motivation behind that is to reduce the complexity of our starting implementation and also make a more comprehensive and portable toolset that focus on the standardization of space OS Testing. Of course our future work mentions how this framework could be adjusted in order for specific use cases and improvements can be integrated in the starting framework.

7.2. Open Source Vulnerability Research

Short Title	Description	Product	Affected Versions
Dynamic Loading Risks [8]	RTEMS enables dynamic loading of executables without proper security checks. An attacker could install high-priority dummy tasks, compromise availability, execute malicious code, and install backdoors.	RTEMS	≤ 5.3
Memory Protection Absence [8]	In RTEMS, the absence of memory protections could allow attackers to execute arbitrary commands and persist on the target.	RTEMS	≤ 5.3
Shell Privilege Escalation [8]	The RTEMS Shell provides development and debugging features but can be exploited for privilege escalation. Users can access it via serial ports or network sockets, potentially obtaining root access.	RTEMS	≤ 5.3
In-Memory File System Exploit [8]	RTEMS default configurations allow users to exploit in-memory file systems (ImFS), enabling data manipulation under an acting task.	RTEMS	≤ 5.3

Table 7.1: Open Source Vulnerabilities for RTEMS real time operating system. When version of the software is not explicitly notated we look at publication date and relative versions available at this point in time, as a means to approximate the possible version of the software used for the evaluation of the findings.

Short Title	Description	Product	Affected Versions
Heap Memory Bounds Issue [32]	Insufficient bounds checking in heap memory management in FreeRTOS before 10.4.3 could lead to undefined behavior.	FreeRTOS	<10.4.3
Stream Buffer Overflow [31]	Integer overflow in `stream_buffer.c` in FreeRTOS before 10.4.3 could lead to buffer overflow vulnerabilities.	FreeRTOS	<10.4.3
Queue Creation Overflow [30]	Integer overflow in `queue.c` for queue creation in FreeRTOS before 10.4.3 could lead to undefined behavior.	FreeRTOS	<10.4.3
Control Data Leak Attack [53]	An external attacker with that can send messages using the Internal Communication Protocol could exploit a heap memory overead vulnerability to achieve a "Control Data Leak" attack. This happens because the function of the command scheduler, the FreeRTOS component which executes the associated command using the included parameters, does not validate the length of the arguments.The researchers mention that the attack was tested in their own ETS-CUBE-1 lab set up.	FreeRTOS	N/A

Table 7.2: Open Source Vulnerabilities for FreeRTOS. When version of the software is not explicitly notated we look at publication date and relative versions available at this point in time, as a means to approximate the possible version of the software used for the evaluation of the findings.

Security vulnerabilities in open-source software have been a major focus of research, as these systems often form the backbone of modern applications, including those in space environments. In this section,

we examine notable vulnerabilities reported in Real-Time Operating Systems (RTOS), hypervisors, and other critical onboard software. Table 7.1 and Table 7.2 summarizes vulnerabilities identified in RTEMS and FreeRTOS, all of which have been studied in various open-source research initiatives. These vulnerabilities provide a lens through which we can assess the risks to onboard space systems and highlight the urgent need for standardized testing frameworks.

7.2.1. Insights from Open-Source Vulnerabilities

Open-source RTOS such as RTEMS and FreeRTOS have been widely deployed in embedded systems, including CubeSats and larger spacecraft [15, 34]. Research into these systems has uncovered a range of vulnerabilities that could be exploited in space environments.

The main consideration for RTEMS are covered in Table 7.1 Dynamic loading risks and lack of memory protection in RTEMS pose a significant threat to system availability and integrity. An attacker with physical or remote access could inject high-priority tasks or execute malicious code, potentially compromising critical mission operations. The RTEMS shell and in-memory file systems (ImFS) could also be leveraged by adversaries to escalate privileges or manipulate onboard data, threatening the confidentiality and availability of mission-critical information.

As displayed in Table 7.2, FreeRTOS presents issues such as heap memory bounds violations, stream buffer overflows, and queue creation vulnerabilities could be exploited to trigger undefined behavior, leading to potential denial-of-service (DoS) attacks or privilege escalation. Notably, the "Control Data Leak" attack in FreeRTOS highlights how an attacker could manipulate inter-process communication, exposing sensitive information or disrupting operations.

7.2.2. Implications for the Space Environment

While many of these vulnerabilities originate in terrestrial applications, their consequences are magnified in space systems due to the unique constraints of the environment. Spacecraft rely heavily on onboard software to manage critical functions such as attitude control, power distribution, and communication. Exploitation of these vulnerabilities could have catastrophic outcomes, including mission failure, interference with multi-tenant systems, or loss of control.

Manipulating onboard software to disrupt operations could render a spacecraft unable to complete its mission or communicate with ground control. Privilege escalation attacks or unauthorized access to debugging tools could allow attackers to seize control of the current execution environment and even try to propagate effect on the rest of the spacecraft, potentially resulting in high-impact actions such as jamming communications or redirecting orbital paths.

The constrained nature of space systems further exacerbates the impact of these vulnerabilities. Limited computational resources, the difficulty of patching or updating software in orbit, and the inability to perform physical interventions make it imperative to identify and mitigate vulnerabilities pre-deployment.

7.2.3. Standardized Testing Frameworks

The vulnerabilities discussed above underscore the pressing need for a standardized approach to testing onboard software and space operating systems. Unlike traditional IT systems, space systems operate in highly adversarial environments, both in terms of physical constraints and potential threats.

A unified framework would ensure comprehensive testing of onboard software against known vulnerabilities and attack vectors. It would facilitate the sharing of security best practices across the aerospace community, enabling collaborative improvement. Moreover, it would streamline the certification process for space software, providing confidence in its robustness before launch. Such a framework could also address specific challenges unique to the space domain, such as radiation-induced faults and real-time performance constraints. In our work, we propose a version of such a framework by providing an example of defining the Threat Model and Risk Assessment Framework of OBSW testing, then implementing a standardized tool-chain that uses dynamic analysis to discover issues in the space OS and finally trying to automate the manual work necessary for triaging and evaluating the discovered issues. At the same time, we extend and integrate SPACE-SHIELD [12] framework in our pipeline to increase the explainability of our discoveries.

Open-source research provides a strong foundation for this initiative, offering a wealth of insights

into how vulnerabilities manifest and how they might be exploited. By leveraging these findings, the aerospace community can take a proactive stance in securing the next generation of space systems.

8

Conclusion

Summary of this work

The security of on-board space systems is becoming an increasingly critical concern as RTOS are integrated into mission-critical functions. This thesis provides a structured approach to evaluating the security of mixed-criticality space applications, focusing on post-exploitation analysis and lateral movement risks in FreeRTOS-based environments. By leveraging a framework based on fuzzing and automated false-positive/duplicate elimination we have identified potential issues which, after validating in a representative testbed, highlight the lack of task isolation and improper privilege separation in RTOS systems commonly used in space missions.

The central contribution of this work is the definition of a standardized and well-motivated Threat Model and Risk Assessment framework and the example implementation of a framework that automates the vulnerability discovery procedure and sets the foundation for a generalized testing process for SpaceOS using state-of-the-art industry-preferred tools. The risk assessment framework also integrates vulnerability assessment standards such as the CVSS vulnerability scoring framework to contextualize security risks within the space domain. This allows for a more accurate assessment of threats based on operational constraints, mission impact, and attacker capabilities. Our collaboration with ESA during responsible disclosure further ensured that our findings were aligned with real-world space security needs.

Moreover, our research contributes to the SPACE-SHIELD framework by expanding on post-exploitation, execution and impact related techniques which could be used by attackers to further compromise the OBSW.

Despite our contributions, several challenges remain. The limitations of fuzzing RTOS due to lack of debugging support in space testbeds, the difficulty of obtaining high-fidelity crash analysis, and the unique constraints of real-time execution environments highlight the need for more adaptable security testing methodologies.

Future Work

This work represents an initial attempt to standardize the kernel fuzzing process for RTOS platforms employed in satellite missions. It establishes a foundational framework that future research can build upon to enhance the robustness and security of space OS components. One of the primary areas for improvement is the refinement of the harness function to enable more sophisticated fuzzing campaigns. Currently, the syscall selection methodology is based on parameter simplicity and AFL++s capability to perform effective mutations. However, this approach could be expanded by developing a more strategic selection process, emphasizing syscalls with complex state dependencies. Furthermore, enhancing the state mutation logic for objects utilized by these syscalls could uncover additional execution paths in the kernel, thereby increasing both code coverage and the discovery of unique crashes.

An important direction for future work involves integrating state-of-the-art kernel-specific fuzzers such as Syzkaller [19], as well as tools designed to detect race-condition vulnerabilities like Rtkaller [38] and

SFuzz [4], which have demonstrated efficacy in testing RTOS systems analogous to our targets. Additionally, the inclusion of custom grammar-based mutators could guide the mutation of memory objects more effectively, reducing the likelihood of meaningless inputs that lead to early program termination.

A more flexible and generalized porting mechanism for the fuzzing framework would enhance its adaptability across different kernel architectures and possibly different architectures. By streamlining the porting process, the framework could support a broader range of RTOS platforms, thus expanding its applicability to a wider variety of space missions and mixed-criticality environments. Additionally, creating a generic API on the software level that abstracts the details of the underlying architecture could enhance the usability and flexibility of the framework , making it possible to perform standardized fuzzing campaigns in different architectures. Such improvements would not only increase the framework's usability but also enable the user to perform consistent security testing across multiple operational contexts.

Building upon the foundation laid for RTOS fuzzing, we aim to extend the scope of our fuzzing framework to include hypervisors, which play a pivotal role in the security of software running on Onboard Computer (OBC) modules in satellite systems. Hypervisors are integral to ensuring temporal and spatial isolation between mixed-criticality applications, making them a high-value target for attackers seeking to escalate privileges or compromise isolated partitions. Consequently, their robustness must be thoroughly evaluated to guarantee the security of space systems.

In this context, integrating AFL++ or other state-of-the-art fuzzers specifically designed for hypervisor technologies such as Hyper-Cube [37], MundoFuzz [29], and Hyperpill [3], into our existing pipeline would be a logical next step. This approach would facilitate comprehensive testing of space hypervisors for vulnerabilities related to temporal and spatial isolation, as well as software flaws that could enable rogue applications to bypass isolation mechanisms. By doing so, our framework could effectively identify software implementation and design bugs that could be used as attack vectors that lead to privilege escalation and potential compromise of critical satellite components.

Finally, it would be interesting to implement some proposed mitigations discussed in section 6.3 and evaluate their impact on space-systems performance. This effort is crucial for assessing the suitability of each mechanism within the space context, where strict availability, reliability, and hard real-time constraints must be maintained to ensure mission success. A possible approach would involve deploying a series of mitigations, e.g sandboxing, memory isolation techniques and Mandatory Access Control (MAC) mechanisms, to measure their effectiveness against the attack vectors identified in our threat model. Following, we would analyze the performance implications of these security enhancements under real-time workloads typical of satellite missions, aiming to balance robust security with minimal performance degradation. To achieve this, we will perform comprehensive benchmarking using representative hardware and software configurations that mimic real-world satellite environments, measuring metrics such as latency, throughput, and resource utilization. Furthermore, we intend to compare our designs and analysis findings with industry standards like ECSS and SAVOIR to ensure compliance with space mission requirements, ultimately guiding the selection of suitable mitigations that maintain operational efficiency while enhancing system security. In addition extending the capabilities of existing mitigations such as the FreeRTOS-MPU enhancements for supporting missing attack surface points, such as integrating protections for Interrupt Service Routines and peripheral device drivers, is rather important for the completeness of the mitigation. Typically this should be paired with the necessary evaluation of the performance fall when the defenses are enabled.

References

[1] *AddressSanitizer Clang 20.0.0git documentation*. URL: https://clang.llvm.org/docs/AddressSanitizer.html (visited on 12/31/2024).

[2] *Architectures of Onboard Data Systems*. en. URL: https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Onboard_Computers_and_Data_Handling/Architectures_of_Onboard_Data_Systems (visited on 03/29/2025).

[3] Alexander Bulekov. "Hyperpill: Fuzzing for Hypervisor-bugs by Leveraging the Hardware Virtualization Interface". en. In: ().

[4] Libo Chen et al. "SFuzz: Slice-based Fuzzing for Real-Time Operating Systems". en. In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. Los Angeles CA USA: ACM, Nov. 2022, pp. 485–498. ISBN: 978-1-4503-9450-5. DOI: 10.1145/3548606.3559367. URL: https://dl.acm.org/doi/10.1145/3548606.3559367 (visited on 10/05/2024).

[5] *Code Coverage LCOV - Visual Studio Marketplace*. en-us. URL: https://marketplace.visualstudio.com/items?itemName=rherrmannr.code-coverage-lcov (visited on 01/18/2025).

[6] *COTS | Activities Portal*. URL: https://activities.esa.int/cluster/920 (visited on 12/23/2024).

[7] crazy hugsy crazy. *hugsy/gef*. original-date: 2015-03-26T22:25:45Z. Jan. 2025. URL: https://github.com/hugsy/gef (visited on 01/18/2025).

[8] James Curbo and Gregory Falco. "Attack Surface Analysis for Spacecraft Flight Software". en. In: ().

[9] *CVSS v3.0 Specification Document*. en. URL: https://www.first.org/cvss/v3.0/specification-document (visited on 01/15/2025).

[10] *ECSS-E-ST-40C Software (6 March 2009) | European Cooperation for Space Standardization*. URL: https://ecss.nl/standard/ecss-e-st-40c-software-general-requirements/ (visited on 02/07/2025).

[11] *ECSS-Q-ST-80C Rev.1 Software product assurance (15 February 2017) | European Cooperation for Space Standardization*. URL: https://ecss.nl/standard/ecss-q-st-80c-rev-1-software-product-assurance-15-february-2017/ (visited on 02/07/2025).

[12] *ESA SPACE-SHIELD*. URL: https://spaceshield.esa.int/ (visited on 09/24/2024).

[13] Gregory Falco. "Cybersecurity Principles for Space Systems". In: *Journal of Aerospace Information Systems* 16.2 (2019). Publisher: American Institute of Aeronautics and Astronautics _eprint: https://doi.org/10.2514/1.I010693, pp. 61–70. ISSN: 1940-3151. DOI: 10.2514/1.I010693. URL: https://doi.org/10.2514/1.I010693 (visited on 06/08/2024).

[14] Gregory Falco, Arun Viswanathan, and Andrew Santangelo. "CubeSat Security Attack Tree Analysis". In: *2021 IEEE 8th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*. July 2021, pp. 68–76. DOI: 10.1109/SMC-IT51442.2021.00016. URL: https://ieeexplore.ieee.org/document/9697673/?arnumber=9697673 (visited on 10/07/2024).

[15] *FreeRTOS documentation - FreeRTOS*. URL: https://freertos.org/Documentation/00-Overview (visited on 09/20/2024).

[16] *Gcov (Using the GNU Compiler Collection (GCC))*. URL: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html (visited on 12/31/2024).

[17] *GDB: The GNU Project Debugger*. URL: https://sourceware.org/gdb/ (visited on 12/31/2024).

[18] Florian Göhler. "Hacking the Stars: A Fuzzing Based Security Assessment of CubeSat Firmware". en. In: ().

[19] *google/syzkaller*. original-date: 2015-10-12T06:05:05Z. Feb. 2025. URL: https://github.com/google/syzkaller (visited on 02/15/2025).

[20] Monowar Hasan et al. "SoK: Security in Real-Time Systems". en. In: *ACM Computing Surveys* 56.9 (Oct. 2024), pp. 1–31. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3649499. URL: https://dl.acm.org/doi/10.1145/3649499 (visited on 09/24/2024).

[21] Ragib Hasan and Raiful Hasan. "Towards a Threat Model and Security Analysis of Spacecraft Computing Systems". In: *2022 IEEE International Conference on Wireless for Space and Extreme Environments (WiSEE)*. ISSN: 2380-7636. Oct. 2022, pp. 87–92. DOI: 10.1109/WiSEE49342.2022.9926912. URL: https://ieeexplore.ieee.org/document/9926912/?arnumber=9926912 (visited on 12/31/2024).

[22] *https://ecss.nl/wp-content/uploads/2022/04/ECSS-Q-ST-70-40C(8April2022).pdf*. URL: https://ecss.nl/wp-content/uploads/2022/04/ECSS-Q-ST-70-40C(8April2022).pdf (visited on 04/09/2025).

[23] *LCOV Code Coverage - The Document Foundation Wiki*. URL: https://wiki.documentfoundation.org/Development/Lcov (visited on 12/31/2024).

[24] *libcsp/libcsp*. original-date: 2011-10-07T10:35:34Z. Mar. 2025. URL: https://github.com/libcsp/libcsp (visited on 03/12/2025).

[25] *LynxOS | POSIX Real Time Operating System*. en. URL: https://www.lynx.com/products/lynxos-178-do-178c-certified-posix-rtos (visited on 10/07/2024).

[26] Gabriele Marra et al. "On the Feasibility of CubeSats Application Sandboxing for Space Missions". en. In: *Proceedings 2024 Workshop on Security of Space and Satellite Systems*. arXiv:2404.04127 [cs]. 2024. DOI: 10.14722/spacesec.2024.23033. URL: http://arxiv.org/abs/2404.04127 (visited on 09/04/2024).

[27] *Memory Protection Unit (MPU) Support - FreeRTOS*. URL: https://freertos.org/Security/04-FreeRTOS-MPU-memory-protection-unit (visited on 02/05/2025).

[28] *MITRE EMB3D*. en. URL: https://emb3d.mitre.org/ (visited on 02/13/2025).

[29] Cheolwoo Myung, Gwangmu Lee, and Byoungyoung Lee. "MundoFuzz: Hypervisor Fuzzing with Statistical Coverage Testing and Grammar Inference". en. In: ().

[30] *NVD - CVE-2021-31571*. URL: https://nvd.nist.gov/vuln/detail/CVE-2021-31571 (visited on 01/18/2025).

[31] *NVD - CVE-2021-31572*. URL: https://nvd.nist.gov/vuln/detail/CVE-2021-31572 (visited on 01/18/2025).

[32] *NVD - CVE-2021-32020*. URL: https://nvd.nist.gov/vuln/detail/CVE-2021-32020 (visited on 01/18/2025).

[33] Jacob G. Oakley. *Cybersecurity for Space: Protecting the Final Frontier*. en. Berkeley, CA: Apress, 2020. ISBN: 978-1-4842-5731-9 978-1-4842-5732-6. DOI: 10.1007/978-1-4842-5732-6. URL: http://link.springer.com/10.1007/978-1-4842-5732-6 (visited on 09/04/2024).

[34] *RTEMS EDISOFT*. en. URL: https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Software_Systems_Engineering/RTEMS_EDISOFT (visited on 12/22/2024).

[35] *SAVOIR*. URL: https://savoir.estec.esa.int/SAVOIRDocuments.htm (visited on 02/07/2025).

[36] Moritz Schloegel et al. "SoK: Prudent Evaluation Practices for Fuzzing". en. In: *2024 IEEE Symposium on Security and Privacy (SP)*. arXiv:2405.10220 [cs]. May 2024, pp. 1974–1993. DOI: 10.1109/SP54263.2024.00137. URL: http://arxiv.org/abs/2405.10220 (visited on 12/31/2024).

[37] Sergej Schumilo et al. "HYPER-CUBE: High-Dimensional Hypervisor Fuzzing". en. In: *Proceedings 2020 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2020. ISBN: 978-1-891562-61-7. DOI: 10.14722/ndss.2020.23096. URL: https://www.ndss-symposium.org/wp-content/uploads/2020/02/23096.pdf (visited on 01/01/2025).

[38] Yuheng Shen et al. "Rtkaller: State-aware Task Generation for RTOS Fuzzing". en. In: *ACM Transactions on Embedded Computing Systems* 20.5s (Oct. 2021), pp. 1–22. ISSN: 1539-9087, 1558-3465. DOI: 10.1145/3477014. URL: https://dl.acm.org/doi/10.1145/3477014 (visited on 10/09/2024).

[39] Andris Slavinskis et al. “ESTCube-1 in-orbit experience and lessons learned”. In: *IEEE Aerospace and Electronic Systems Magazine* 30.8 (Aug. 2015). Conference Name: IEEE Aerospace and Electronic Systems Magazine, pp. 12–22. ISSN: 1557-959X. DOI: 10.1109/MAES.2015.150034. URL: https://ieeexplore.ieee.org/document/7286959/?arnumber=7286959 (visited on 12/22/2024).

[40] “Space Data Link Security Protocol”. en. In: (2022).

[41] “Space Data Link Security ProtocolExtended Procedures”. en. In: (2020).

[42] “Space engineering”. en. In: (2008).

[43] *SPARTA*. URL: https://sparta.aerospace.org/ (visited on 12/23/2024).

[44] *Standards | European Cooperation for Space Standardization*. URL: https://ecss.nl/standards/ (visited on 12/23/2024).

[45] 24A06-Security for Space Systems (3S) 2024. *Space Systems Security Challenge - 24A06 - Security for Space Systems (3S) 2024 - ESTEC*. URL: https://atpi.eventsair.com/24a06---3s2024/atpi.eventsair.com/space-systems-security-challenge (visited on 03/12/2025).

[46] *The AFL++ fuzzing framework*. en. URL: https://aflplus.plus/ (visited on 09/24/2024).

[47] Shahid Ul Haq et al. “A survey on IoT & embedded device firmware security: architecture, extraction techniques, and vulnerability analysis frameworks”. en. In: *Discover Internet of Things* 3.1 (Oct. 2023), p. 17. ISSN: 2730-7239. DOI: 10.1007/s43926-023-00045-2. URL: https://link.springer.com/10.1007/s43926-023-00045-2 (visited on 09/24/2024).

[48] Serge Valera. “October 2013 ECSS WG Draft”. en. In: ().

[49] *Valgrind Home*. URL: https://valgrind.org/ (visited on 12/31/2024).

[50] Arun Viswanathan et al. “Secure-by-component: A System-of-systems Design Paradigm for Securing Space Missions”. en. In: *2024 Security for Space Systems (3S)*. Noordwijk, Netherlands: IEEE, May 2024, pp. 1–9. ISBN: 978-90-90-38704-8. DOI: 10.23919/3S60530.2024.10592289. URL: https://ieeexplore.ieee.org/document/10592289/ (visited on 09/18/2024).

[51] *VxWorks | Industry Leading RTOS for Embedded Systems*. en. URL: https://www.windriver.com/products/vxworks (visited on 10/07/2024).

[52] Sander Wiebing, Thomas Rooijakkers, and Sebastiaan Tesink. *Improving AFL++ CmpLog: Tackling the bottlenecks*. en. arXiv:2211.08357 [cs]. Nov. 2022. DOI: 10.48550/arXiv.2211.08357. URL: http://arxiv.org/abs/2211.08357 (visited on 01/18/2025).

[53] Johannes Willbold et al. “Space Odyssey: An Experimental Software Security Analysis of Satellites”. en. In: *2023 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2023, pp. 1–19. ISBN: 978-1-66549-336-9. DOI: 10.1109/SP46215.2023.10351029. URL: https://ieeexplore.ieee.org/document/10351029/ (visited on 08/15/2024).

[54] Nikita Yadav et al. “Orbital Shield: Rethinking Satellite Security in the Commercial Off-the-Shelf Era”. en. In: *2024 Security for Space Systems (3S)*. Noordwijk, Netherlands: IEEE, May 2024, pp. 1–11. ISBN: 978-90-90-38704-8. DOI: 10.23919/3S60530.2024.10592292. URL: https://ieeexplore.ieee.org/document/10592292/ (visited on 08/15/2024).

[55] Pingyue Yue et al. *Low Earth Orbit Satellite Security and Reliability: Issues, Solutions, and the Road Ahead*. en. arXiv:2201.03063 [eess]. July 2023. URL: http://arxiv.org/abs/2201.03063 (visited on 09/04/2024).

A

Crash Triaging and Coverage Generation Automation

Here we present our scripts used to automate the fuzzing and output-crash parsing and de-duplication, as well as the script used for yielding coverage information.

Listing A.1: run.sh

```
#!/bin/bash

fuzz_group="all"
TIME_TO_FUZZ=60 #86400 # 24 hours in seconds

# make sure the first argument is one of tasks, queue, semaphore, stream_buffer, message_buffer
if [ "$1" != "tasks" ] && [ "$1" != "queue" ] && [ "$1" != "semaphore" ] && [ "$1" != "stream_buffer" ] && [ "$1" != "message_buffer" ]; then
    echo "Did not specified [tasks|queue|stream_buffer|message_buffer], will fuzz all"
    sleep 1
else
    fuzz_group=$1
    echo "Fuzzing Group: $1"
fi

multi_thread=0
# if second argument is multi_thread, then run with multi_thread label for the fuzz group
if [ "$1" == "multi_thread" ] || [ "$2" == "multi_thread" ]; then
    multi_thread=1
    prefix=multi-thread-
    echo "Multi-threading enabled"
    export AFL_LLVM_THREADSAFE_INST=1
    sleep 1
fi

export AFL_DISABLE_TRIM=1
export CC=$(realpath ./AFLplusplus/afl-clang-fast)
export AFL_LLVM_ALLOWLIST=$(realpath allow-instr.list.afl)

function run_main_afl () {
    # run the fuzzer with no hup
    # then get the PID and check if it failed
    # if failed wait a second and retry.
    # Maximum of 5 retries
    nohup ./afl-fuzz -i inputs/$1 -o report-afl-${prefix}$1 -V $TIME_TO_FUZZ -M $2 $3 -- ./build-$1/kernel-fuzzer > nohup-$1-$2.out 2>&1 &

    PID=$!

```

```
38      for i in {1..5}; do
39          if ! kill -0 $PID; then
40              echo "Fuzzer␣failed␣to␣start,␣retrying..."
41              sleep 1
42              nohup ./afl-fuzz -i inputs/$1 -o report-afl-${prefix}$1 -V $TIME_TO_FUZZ -M
                    $2 $3 -- ./build-$1/kernel-fuzzer > nohup-$1-$2.out 2>&1 &
43              PID=$!
44          else
45              break
46          fi
47      done
48
49  }
50
51  function run_sec_afl () {
52      nohup ./afl-fuzz -i inputs/$1 -o report-afl-${prefix}$1 -V $TIME_TO_FUZZ -S $2 $3
            -- ./build-$1/kernel-fuzzer > nohup-$1-$2.out 2>&1 &
53
54      PID=$!
55
56      for i in {1..5}; do
57          if ! kill -0 $PID; then
58              echo "Fuzzer␣failed␣to␣start,␣retrying..."
59              sleep 1
60              nohup ./afl-fuzz -i inputs/$1 -o report-afl-${prefix}$1 -V $TIME_TO_FUZZ -S
                    $2 $3 -- ./build-$1/kernel-fuzzer > nohup-$1-$2.out 2>&1 &
61              PID=$!
62          else
63              break
64          fi
65      done
66  }
67
68  function afl_w_san() {
69      export AFL_USE_ASAN=1 \
70      && compile_trg \
71      && run_sec_afl $fuzz_group fuzzer-sanitizers-asan-1 "␣-P␣explore␣-a␣binary␣-p␣
            exploit" \
72      && run_sec_afl $fuzz_group fuzzer-sanitizers-asan-2 "␣-P␣exploit␣-p␣explore" \
73      && run_sec_afl $fuzz_group fuzzer-sanitizers-asan-3 "␣-P␣explore␣-p␣fast"
74      unset AFL_USE_ASAN
75
76      export AFL_USE_UBSAN=1 \
77      && compile_trg \
78      && run_sec_afl $fuzz_group fuzzer-sanitizers-ubsan-1 "␣-P␣explore␣-a␣binary␣-p␣
            exploit" \
79      && run_sec_afl $fuzz_group fuzzer-sanitizers-ubsan-2 "␣-P␣exploit␣-p␣explore" \
80      && run_sec_afl $fuzz_group fuzzer-sanitizers-ubsan-3 "␣-P␣explore␣-p␣fast"
81      unset AFL_USE_UBSAN
82
83      # this does not work
84      # export AFL_USE_TSAN=1
85      # compile_trg
86      # run_sec_afl $fuzz_group fuzzer-sanitizers-tsan-1 " -P explore -a binary -p
            exploit"
87      # run_sec_afl $fuzz_group fuzzer-sanitizers-tsan-2 " -P exploit -p explore"
88      # unset AFL_USE_TSAN
89
90  }
91
92  function afl_cmplog() {
93      export AFL_LLVM_CMPLOG=1
94
95      compile_trg \
96      && run_main_afl $fuzz_group fuzzer-cmplog-1 "-a␣binary␣-p␣fast␣-P␣explore" \
97      && run_sec_afl $fuzz_group fuzzer-cmplog-2 "-l1AT␣-P␣explore␣-p␣explore" \
98      && run_sec_afl $fuzz_group fuzzer-cmplog-3 "-Z␣-l1ATX␣-P␣explore␣-p␣quad␣-P␣exploit
            "
99
100     unset AFL_LLVM_CMPLOG
101 }
```

```

function compile_trg {
    mkdir -p build-${fuzz_group} && \
    cd build-$fuzz_group && \
    cmake -DFUZZ_GROUP=$fuzz_group -DMULTI_THREAD=$multi_thread .. && \
    make clean all && \

    cd ..
}


afl_cmplog \
&& afl_w_san

# if IS_DOCKER=1, then this command
if [ "$IS_DOCKER" == "1" ]; then
    tail -f nohup-$fuzz_group-*.out
fi


# run the fuzzer for the testcase of choice and redirect to nohup-$fuzz_group.out

# ./afl-fuzz -i inputs/$fuzz_group -o report-afl-$fuzz_group -V $TIME_TO_FUZZ -S
    fuzzer01 -- ./build-$fuzz_group/kernel-fuzzer &
# ./afl-fuzz -i inputs/$fuzz_group -o report-afl-$fuzz_group -V $TIME_TO_FUZZ -S
    fuzzer02 -- ./build-$fuzz_group/kernel-fuzzer;
# nohup ./afl-fuzz -i inputs/$1 -o report-afl-$1 -V $TIME_TO_FUZZ -- ./build-$1/kernel-
    fuzzer > nohup-$1.out 2>&1 & echo "Running $1 fuzzing...";
```

Listing A.2: run-crashes.sh

```
#!/bin/bash

# check arguments
if [ "$1" != "tasks" ] && [ "$1" != "queue" ] && [ "$1" != "semaphore" ] && [ "$1" != "
    stream_buffer" ] && [ "$1" != "message_buffer" ] && [ "$1" != "all" ]; then
    echo "Did␣not␣specified␣[tasks|queue|stream_buffer|message_buffer]"
    exit 1
else
    fuzz_group=$1
    echo "Syscall␣Group:␣$1"
fi

multi_thread=0
# if second argument is multi_thread, then run with multi_thread label for the fuzz
    group
if [ "$2" == "multi_thread" ]; then
    multi_thread=1
    echo "Multi-threading␣enabled"
fi

# check if the third argument is a directory
if [ -d "$3" ]; then
    echo "Using␣AFL␣report␣directory:␣$3"
    afl_report_dir=$3
else
    echo "Invalid␣AFL␣report␣directory"
    exit 1
fi

function compile_trg() {
    debug=$1
    asan_sup=$2
    # compile the program with coverage flags
    rm -rf build-$fuzz_group
    mkdir -p build-$fuzz_group && \
    cd build-$fuzz_group && \
    cmake -DFUZZ_GROUP=$fuzz_group -DMULTI_THREAD=$multi_thread -DDEBUG=$debug -DASAN=
        $asan_sup .. && \
    make clean all && \
    cd ..
```

```
}

echo "Retrieving crashes and hangs from $afl_report_dir"

# Get all the input files in the 'hang' and 'crashes' directory
# hangfiles=$(find $afl_report_dir/*/hangs -iname "id*")
crashes_tc=$(find $afl_report_dir/*/crashes -iname "id*")

# print the number of crashes and hangs
echo "Number of crashes retrieved: $(echo $crashes_tc | wc -w)"

# create a temporary directory to store the output
tmp_dir=$(mktemp -d -p ./)

mkdir -p $tmp_dir/hangs
mkdir -p $tmp_dir/crashes
mkdir -p $tmp_dir/other

# just compile the program
compile_trg 0 0

# run the program for all the crashes and according to the return code determine if the crashes cause a seg fault or a hang
for f in $crashes_tc; do
    # run the program with timeout of 10 milliseconds
    (timeout --signal=SIGINT 1 ./build-$fuzz_group/kernel-fuzzer < $f) &> /dev/null
    if [ $? -eq 139 ]; then
        cp $f $tmp_dir/crashes/
    elif [ $? -eq 124 ]; then
        cp $f $tmp_dir/hangs/
    else
        cp $f $tmp_dir/other/
    fi
done

echo "$(ls $tmp_dir/crashes/ | wc -w) crashes are stored in $tmp_dir/crashes"
echo "$(ls $tmp_dir/hangs/ | wc -w) hangs are stored in $tmp_dir/hangs"
echo "$(ls $tmp_dir/other/ | wc -w) others are stored in $tmp_dir/hangs"

mkdir -p $tmp_dir/crashes/out
mkdir -p $tmp_dir/crashes/out/gdb
mkdir -p $tmp_dir/crashes/out/valgrind
mkdir -p $tmp_dir/crashes/out/asan

# compile for debug first
compile_trg 1 0

crashes=$(find $tmp_dir/crashes -iname "id*")

# run the program for all the crashes and store the output in a file
for f in $crashes; do
    # run the program with timeout of 2s
    echo "Running $f"
    out_file_gdb=$tmp_dir/crashes/out/gdb/$(basename $f)
    out_file_valgrind=$tmp_dir/crashes/out/valgrind/$(basename $f)
    echo "r < $f" | timeout --signal=SIGINT 2 gdb ./build-$fuzz_group/kernel-fuzzer > $out_file_gdb 2>&1
    timeout --signal=SIGINT 2 valgrind --log-file=$out_file_valgrind ./build-$fuzz_group/kernel-fuzzer < $f
done

# Now compiling with ASAN support
compile_trg 1 1
# run the program for all the crashes and store the output in a file
for f in $crashes; do
    # run the program with timeout of 2s
    echo "Running $f"
    out_file_asan=$tmp_dir/crashes/out/asan/$(basename $f)
    timeout --signal=SIGINT 2 ./build-$fuzz_group/kernel-fuzzer < $f > $out_file_asan`` 2>&1
done
```

```
105
106 [[ -d crash-runs-prev ]] && rm -rf crash-runs-prev
107 [[ -d crash-runs ]] && mv crash-runs{,-prev}
108 mv $tmp_dir crash-runs
```

Listing A.3: unique-crashes.sh

```
1  #!/bin/bash
2
3  # This script will take a directory of crash files, and output a list of unique crashes
4
5  USAGE="Usage: $0 <crash_dir> [asan|gdb|valgrind]"
6
7  # First argument is the crash directory, check and set
8  if [ -z "$1" ] || ! [ -d "$1" ]; then
9      echo $USAGE
10     exit 1
11 fi
12
13 # check if the word "valgrind" is in cmd args
14 if [[ "$@" == *valgrind* ]]; then
15     echo "Valgrind detected"
16     CHECK_VALGRIND=1
17 else
18     CHECK_VALGRIND=0
19 fi
20
21 if [[ "$@" == *asan* ]]; then
22     echo "asan detected"
23     CHECK_ASAN=1
24 else
25     CHECK_ASAN=0
26 fi
27
28 if [[ "$@" == *gdb* ]]; then
29     echo "gdb detected"
30     CHECK_GDB=1
31 else
32     CHECK_GDB=0
33 fi
34
35 if [ $CHECK_VALGRIND -eq 0 ] && [ $CHECK_ASAN -eq 0 ] && [ $CHECK_GDB -eq 0 ]; then
36     echo "No tool specified to check for unique crashes, will use all"
37     CHECK_VALGRIND=1
38     CHECK_ASAN=1
39     CHECK_GDB=1
40 fi
41
42 CRASH_DIR=$1
43
44 crash_files=$(find $CRASH_DIR -maxdepth 1 -type f -iname "id*")
45 asan_files=$(find $CRASH_DIR/out/asan -type f -iname "id*")
46 gdb_files=$(find $CRASH_DIR/out/gdb -type f -iname "id*")
47 valgrind_files=$(find $CRASH_DIR/out/valgrind -type f -iname "id*")
48
49 # get the hashes of each crash file for each tool
50 crash_hashes=$(for file in $crash_files; do md5sum $file; done | awk '{print $1}')
51 asan_hashes=$(for file in $asan_files; do md5sum $file; done | awk '{print $1}')
52 gdb_hashes=$(for file in $gdb_files; do md5sum $file; done | awk '{print $1}')
53 valgrind_hashes=$(for file in $valgrind_files; do cat $file | grep 'by ' | awk -F ' ' '
       {print $4}' | md5sum; done | awk '{print $1}')
54
55 # create a list of tuples of the file name and all of its hashes
56 crash_files_hashed=$(paste -d "~" <(echo "$crash_files") <(echo "$crash_hashes") <(echo
        "$asan_hashes") <(echo "$gdb_hashes") <(echo "$valgrind_hashes") | awk '{print $1,
       $2,$3,$4,$5}')
57
58 unique_files=""
59
60 for tuple in $crash_files_hashed; do
61     hash=$(echo $tuple | cut -d "~" -f 2)
```

```
62      asan_hash=$(echo $tuple | cut -d "~" -f 3)
63      gdb_hash=$(echo $tuple | cut -d "~" -f 4)
64      valgrind_hash=$(echo $tuple | cut -d "~" -f 5)
65
66      # print all the unique files and grep for each one of the hashes. If none is found
            then include the file in the unique list
67      if ! echo $unique_files | grep -q $hash ; then
68          if [ $CHECK_ASAN -eq 0 ] || ! echo $unique_files | grep -q $asan_hash ; then
69              if [ $CHECK_GDB -eq 0 ] ||  ! echo $unique_files | grep -q $gdb_hash ; then
70                  if [ $CHECK_VALGRIND -eq 0 ] ||  ! echo $unique_files | grep -q
                        $valgrind_hash ; then
71                      unique_files="$unique_files $tuple"
72                  fi
73              fi
74          fi
75      fi
76  done
77  tmp1=($(echo $unique_files))
78  tmp2=($(echo $crash_files))
79  # print how many unique files were found out of the total
80  echo "Found ${#tmp1[@]} unique crashes out of ${#tmp2[@]} total crashes"
81
82  mkdir -p $CRASH_DIR/unique
83  mkdir -p $CRASH_DIR/unique/out
84  mkdir -p $CRASH_DIR/unique/out/asan
85  mkdir -p $CRASH_DIR/unique/out/gdb
86  mkdir -p $CRASH_DIR/unique/out/valgrind
87
88  # populate the unique directory with the unique files
89  for file in $unique_files; do
90      filename=$(echo $file | cut -d "~" -f 1 | xargs basename)
91      # make sure that the file name exists in all source dirs
92      if [ ! -f $CRASH_DIR/$filename ] || [ ! -f $CRASH_DIR/out/asan/$filename ] || [ ! -
            f $CRASH_DIR/out/gdb/$filename ] || [ ! -f $CRASH_DIR/out/valgrind/$filename ];
             then
93          echo "File $filename not found in all source directories"
94          continue
95      fi
96
97      cp  $CRASH_DIR/$filename $CRASH_DIR/unique/$filename
98      cp  $CRASH_DIR/out/asan/$filename $CRASH_DIR/unique/out/asan/$filename
99      cp  $CRASH_DIR/out/gdb/$filename $CRASH_DIR/unique/out/gdb/$filename
100     cp  $CRASH_DIR/out/valgrind/$filename $CRASH_DIR/unique/out/valgrind/$filename
101
102     # check that the copies where successful
103     if [ ! -f $CRASH_DIR/unique/$filename ] || [ ! -f $CRASH_DIR/unique/out/asan/
            $filename ] || [ ! -f $CRASH_DIR/unique/out/gdb/$filename ] || [ ! -f
            $CRASH_DIR/unique/out/valgrind/$filename ]; then
104         echo "Failed to copy $filename to unique directory"
105     fi
106 done
107
108 # print how many files in each directory
109 echo "Unique crashes:"
110 echo "ASAN: $(ls $CRASH_DIR/unique/out/asan | wc -l)"
111 echo "GDB: $(ls $CRASH_DIR/unique/out/gdb | wc -l)"
112 echo "Valgrind: $(ls $CRASH_DIR/unique/out/valgrind | wc -l)"
```

Listing A.4: run-coverage.sh

```
1
2   #!/bin/bash
3
4   # Usage: ./run_coverage.sh [tasks|queue|semaphore|stream_buffer|message_buffer|all] [
        multi_thread] <afl-report-dir>
5
6   # check arguments
7   if [ "$1" != "tasks" ] && [ "$1" != "queue" ] && [ "$1" != "semaphore" ] && [ "$1" != "
        stream_buffer" ] && [ "$1" != "message_buffer" ] && [ "$1" != "all" ]; then
8       echo "Did not specified [tasks|queue|stream_buffer|message_buffer]"
9       exit 1
```

```
10 else
11     fuzz_group=$1
12     echo "Syscall␣Group:␣$1"
13 fi
14
15 multi_thread=0
16 # if second argument is multi_thread, then run with multi_thread label for the fuzz
       group
17 if [ "$2" == "multi_thread" ]; then
18     multi_thread=1
19     echo "Multi-threading␣enabled"
20 fi
21
22 # check if the third argument is a directory
23 if [ -d "$3" ]; then
24     echo "Using␣AFL␣report␣directory:␣$3"
25     afl_report_dir=$3
26 else
27     echo "Invalid␣AFL␣report␣directory"
28     exit 1
29 fi
30
31 # the input files are in the input directory
32 input_corpus=$(find $afl_report_dir -iname "id*")
33
34
35 # make sure
36 # export CC=$(which clang-14)
37
38 # compile the program with coverage flags
39 mkdir -p build-$fuzz_group && \
40 cd build-$fuzz_group && \
41 cmake -DFUZZ_GROUP=$fuzz_group -DMULTI_THREAD=$multi_thread -DCOVERAGE=1 .. && \
42 make clean all && \
43
44 cd ..
45
46 for f in $input_corpus; do
47     # run the program with timeout of 10 milliseconds
48     echo "Running␣$f"
49     (timeout --signal=SIGINT 1 ./build-$fuzz_group/kernel-fuzzer < $f) &> /dev/null
50 done
51
52
53 tmp_dir=$(mktemp -d -p ./)
54 # run lcov to generate coverage report
55 lcov -d ./build-$fuzz_group --capture -o $tmp_dir/coverage.info
56 lcov -r $tmp_dir/coverage.info '/usr/include/*' -o $tmp_dir/filtered.info.lcov --ignore
       -errors unused
57 genhtml -o coverage-html $tmp_dir/filtered.info.lcov --legend
58
59 mv $tmp_dir coverage-lcov
60
61 echo "Coverage␣report␣is␣generated␣in␣coverage-html/index.html"
62
63 # open up the coverage report in the browser
64 # xdg-open coverage-html/index.html
```