UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

SYSTEMATIC FUZZ TESTING TECHNIQUES ON A NANOSATELLITE FLIGHT SOFTWARE FOR AGILE MISSION DEVELOPMENT

TESIS PARA OPTAR AL GRADO DE MAGISTER EN
CIENCIAS, MENCIÓN COMPUTACIÓN
MEMORIA PARA OPTAR AL TITULO DE INGENIERA CIVIL EN COMPUTACIÓN

TAMARA GUTIÉRREZ ROJO

PROFESOR GUÍA:
ALEXANDRE BERGEL

MIEMBROS DE LA COMISIÓN:
MARÍA CECILIA BASTARRICA PIÑEYRO
MARCOS DÍAZ QUEZADA
MIGUEL CAMPUSANO ARAYA

Esta trabajo ha sido parcialmente financiado por:
ANID Fondecyt Regular 1200067

SANTIAGO DE CHILE
2022

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE: Ingeniera Civil en Computación y
grado de Magíster en Ciencias, mención Computación
POR: Tamara Gutiérrez Rojo
FECHA: 2022
PROF. GUÍA: Alexandre Bergel

TÉCNICAS SISTEMÁTICAS DE FUZZ TESTING APLICADO AL SOFTWARE DE VUELO DE UN NANOSATÉLITE PARA UN DESARROLLO DE MISIÓN ÁGIL

El éxito de las misiones de CubeSats depende de su rendimiento en un ambiente extremo. El software de vuelo es un componente crítico que maneja todas estas operaciones. La literatura muestra que las misiones de CubeSats sufren una alta mortalidad infantil, y muchas de las fallas de las naves espaciales están relacionadas a errores de software de vuelo, algunas de ellas resultando en un fracaso total de la misión. Mientras otras áreas incluyen técnicas de testing de software avanzadas, las soluciones de software para CubeSats dependen mayoritariamente de testing unitario, software in the loop simulation y hardware in the loop simulation. Sin embargo, los requerimientos de “El Nuevo Espacio” presionan para añadir “agilidad” al desarrollo, lo que podría limitar la capacidad de testing. En este trabajo, técnicas de fuzz testing fueron desarrolladas, implementadas y evaluadas para facilitar el testing operacional de software de vuelo de CubeSats, a la vez que se mantiene su robustez. El impacto de las herramientas fue evaluado en tres nuevos CubeSats en desarrollo, en la Universidad de Chile. La aplicación identificó doce nuevos bugs en menos de tres días. Estas fallas fueron reportadas, reparadas y caracterizadas en ocho sesiones de sprint. Los resultados indican que el fuzz testing mejoró la completitud de testing de software de vuelo a través de la automatización y con casi ninguna interrupción en el desarrollo. Esta solución también es aplicable a arquitecturas, y a otros sistemas que siguen una arquitectura similar.

SYSTEMATIC FUZZ TESTING TECHNIQUES ON A NANOSATELLITE FLIGHT SOFTWARE FOR AGILE MISSION DEVELOPMENT

The success of CubeSat missions depends on their performance in a harsh environment. Flight software is a critical component that manages all of these operations. Literature shows that CubeSat missions suffer high infant mortality, and many spacecraft failures are related to flight software errors, some of them resulting in complete mission loss. While other fields include advanced software testing techniques, CubeSat software solutions mostly rely on unit testing, software in the loop simulation, and hardware in the loop simulation. However, the "New Space" requirements pressure to add "agility" to the development, which could limit the capacity to test. In this work, fuzz testing techniques were developed, implemented, and evaluated to expedite operational testing of CubeSats' flight software while maintaining its robustness. The impact of the tools was evaluated in three new CubeSats under development at the University of Chile. The application identified twelve new bugs in less than three days. These failures were reported, fixed, and characterized in eight sprint sessions. The results indicate that fuzz testing improved the completeness of flight software testing through automation and with almost no development interruption. This solution is also applicable to other architectures, and other systems that follow a similar architecture.

Dedicado a mis padres,
por su amor y apoyo incondicional

Acknowledgments

En primer lugar, quisiera agradecer a mi familia por todo el apoyo dado durante este largo camino que he decidido seguir. Mis padres, en lo particular, han sido uno de los grandes pilares y quienes me han enseñado que el esfuerzo, la perserverencia y el amor por lo que hacemos día a día es más importante que cualquier otro recurso. Sin ellos no hubiese tenido la capacidad de volver a levantar luego de mis caídas. Mi mayor dedicación y reconocimiento es para ellos, quienes se sobresforzaron durante muchísimos años por mí, mi salud y mi educación dentro de un sistema que muchas veces no es justo. Los quiero desde aquí al infinito.

En segundo lugar, le agradezco a mi novio, Carlos, quien siempre me estuvo apoyando desde su bondad, experiencia y transparencia. Eres un gran compañero de vida, y quien también me ha enseñado que la fortaleza es algo que se puede adquirir a través de las experiencias y las decisiones que uno está dispuesto a tomar. Sin duda siempre me has apoyado y aconsejado honestamente, lo cual te agradeceré toda mi vida porque todas tus enseñanzas son parte importante de lo que hoy soy. Gracias por tu paciencia, dedicación y preocupación.

También quiero agradecer a mi profesor guía, Alexandre Bergel, quien tuvo la paciencia y la disposición de guiar mi trabajo. No tengo dudas de que es un gran profesor y motivador, con una vasta experiencia. Me siento muy agradecida de haber tenido la oportunidad de haber trabajado con él porque me enseñó, entre tantas cosas, lo importante que es la consistencia, el detalle y la perseverancia en un trabajo de esta índole. Así descubrí lo enriquecedor que fue este proceso de investigación. Muchas gracias por darse el tiempo de tener reuniones semanales, eso es algo que no hacen todos los profesores realmente. Por último, quería darle las gracias por ser una de las personas que me instó a conseguir un puesto dentro de una agencia espacial, un sueño que tenía desde niña, pero que no sabía si se podría cumplir. Gracias a usted fue posible.

El equipo de SPEL también fue quién me permitió poder desarrollarme en esta área desde 2018, y quien me abrió las puertas a descubrir lo que es la investigación. El profesor Marcos Díaz y todos los compañeros del laboratorio forman un gran grupo de trabajo. Quería agradecerles a todos ellos por la oportunidad que me dieron de aportar un granito de arena a este proyecto, el cual creo que merece mucho más reconocimiento y apoyo del que actualmente tiene. Algunos de ustedes no lo saben, pero durante la segunda mitad de mi paso por la universidad fueron el único pilar donde pude autodescubrirme en términos profesionales y también personales.

Mis agradecimientos también van hacia todos mis amigos y familia en general. Específicamente, gracias a mis amigos del alma, Juan Pablo, Valentina y Felipe. Son tantos los años que nos unen y siempre han estado en mi vida aunque estemos separados.

Por último quiero dar un especial reconocimiento a Pacu, mi perrito senior analytics, que aunque no es completamente mío, siempre estuvo apoyándome con sus cariños, lengüetazos y las tantas cosas que sabe en su mente perruna. Gracias por hacerme la vida más bella y amena.

Contents

Índice de Tablas

Índice de Ilustraciones

Chapter 1

Introduction

1.1. Motivation

The first conception of a CubeSat nanosatellite prototype, a class of nanosatellites with standard size and form factor, came up only 20 years ago approximately. Initially, CubeSats were conceived with a mainly educational purpose in which students can experience the development and operation of a satellite in the time frame of a college degree [1]. Later, they became an effective tool for making scientific discoveries and developing new technologies to improve diverse processes. Nowadays, nanosatellites have opened several opportunities, but they still need to overcome multiple challenges to reach their full potential [2]. Due to this recent expansion, nanosatellites increasingly require more attention to their quality attributes to succeed in complex missions. Specifically, flight software of nanosatellites is critical in determining a satellite's quality because it controls most of the tasks that must be executed once in orbit. The success rate of space missions is highly dependent on the quality of the flight software [3].

Several testing techniques are used to assess flight software quality in the space field. However, the most advanced techniques are only suitable for larger missions or systems, in terms of time and budget, such as large satellites, rovers, or interplanetary missions [4]. In the current literature, the most reported testing techniques applied to nanosatellites' flight software testing are hardware in the loop simulation (HILS), and software in the loop simulation (SILS) [5, 6]. HILS and SILS methodologies can optimize the production process' overall costs in certain situations [7, 8]. However, these techniques can be difficult to implement and execute, potentially dangerous to the hardware when executed in engineering or flight models, and time-consuming to set up the environment. Besides, test cases must be predefined because these techniques are challenging to automate [7].

In a recent review of some relevant nanosatellite flight software frameworks, only three out of six candidates exhibit the reliability attribute, which refers to the existence of unit testing with significant code coverage [9]. Implementing different testing techniques also relies on the flight software design. Satellite command and data handling (C&DH) systems are usually designed to receive telecommands, execute necessary actions, and answer with data obtained from telemetry. Some novel flight software designs exploit this concept to implement a command-based software architecture [10, 11]. Such a clear design and well-documented interfaces may help implement testing strategies that treat the flight software as a black-box instead of intervening the code with unit testing or instrumentation.

1.2. Problem Statement

The papers of this area barely describe the testing systems applied to the flight software of CubeSats. The approaches found mention unit testing, HILS techniques, or software tools that facilitate the data and command handling from the ground station. However, they do not consider automated testing techniques that could be useful to optimize the working time of CubeSat projects, which is one of the most common problems for flight software development, especially in small groups studying and developing new technologies in this field.

SUCHAI I was the first CubeSat mission developed in Chile in the Space and Planetary Exploration Laboratory (SPEL) at the University of Chile. The context under which this mission was developed is merely academic. The team comprises researchers, engineers, and students from different areas. However, the group is small and a pioneer in developing nanosatellites in Chile. Given its characteristics, the team has adopted an agile methodology to develop SUCHAI I and its current missions, SUCHAI II, III, and Plantsat.

The SUCHAI flight software is a fundamental part of the SUCHAI series of nanosatellites. The flight software of the different nanosatellite missions is being developed under a project in a VCS repository. The team has implemented several testing techniques to assure the quality of this flight software, such as unit testing, integration testing, and HILS. However, unit testing and integration testing require time to cover a considerable amount of cases, and HILS needs time to set up the environment, and it is not trivial to automate.

In need of looking for an agile testing technique that can improve the quality assurance of nanosatellites space missions and can follow their agile development requirements, the main aim of this work is to implement, apply and analyze fuzz testing strategies in the SUCHAI flight software, uncovering vulnerabilities that previous techniques have not found. The searched failures are related to the availability and reliability of the system, associated explicitly with software crashes [12].

Fuzz testing is an automated software testing technique that consists in automatically generating random input to find software vulnerabilities [13]. Thanks to the design of the SUCHAI flight software, the software can be intervened by sending commands and observing its behavior. Therefore, fuzz testing is implemented by generating a set of random commands and parameters. The randomness of the number of commands, the number of parameters, the commands characters composition, and the parameters character composition give rise to four proposed strategies defined in Section 4.2. The bugs found are later being reported to the SUCHAI software team and characterized, giving highlights to apply this technique in other missions.

1.3. Research Questions

- Which testing techniques can be applied to the SUCHAI flight software? How can they be integrated into the SUCHAI flight software?

- What are the advantages and disadvantages of fuzz testing compared to other testing techniques introduced in the nanosatellite missions context?

- Is it possible to find new failures in the SUCHAI flight software by applying fuzz testing techniques that the previous testing methodologies could not find before? If that is the case, how can those failures be characterized?

1.4. Hypothesis

The application of fuzz testing strategies will find new runtime failures in the SUCHAI flight software.

1.5. Objectives

1.5.1. General Objectives

The general objective of this work is to identify and characterize software vulnerabilities in the SUCHAI flight software using fuzz testing strategies, improving the flight software quality and speeding up the testing process.

1.5.2. Specific Objectives

- Explore the different fuzz testing techniques explained in "The Fuzzing Book" (Zeller *et al.*, 2019) [14].
- Explore the advantages and disadvantages of the current testing practices carried out in the SUCHAI flight software.
- Create and develop different fuzz testing strategies to detect failures involved with crashes.
- Test the different strategies on the SUCHAI flight software and capture results related to return codes, memory consumption, built-in assertions, among others.
- Generate data visualization and statistics that help compare the fuzzing strategies implemented on the SUCHAI flight software.
- Report the bugs found to the SPEL team and suggest possible solutions to improve the testing system of the SUCHAI flight software.
- Monitor fixes made by the SPEL team to address the identified software issues.
- Characterize the failures found.
- Analyze possible applications of this technique to other nanosatellite flight software.

1.6. Contributions

This thesis presents the impact of using fuzz testing to verify the proper flight software operation of nanosatellites. The evaluation was performed in a series of 3 nanosatellites being developed at the University of Chile (SUCHAI-II, SUCHAI-III, and PlantSat). The contributions made by this work are:

- Presents a methodology to apply fuzz testing to nanosatellites' flight software as part of an agile CubeSat flight software methodology;

- Highlights and discusses the challenges faced and describes the main requirements to implement this technique in similar projects;
- Presents a compelling case study of applying a modern testing technique to critical embedded software, which opens a niche in the field of nanosatellite flight software testing.

As a result, fuzz testing has proven to be very valuable in this context as:

(i) various potential software failures whose severity ranged from middle to severe were discovered

(ii) a sequence of commands to trigger and reproduce each of these failures was identified, and

(iii) these software failures were addressed.

1.7. Publications

1.7.1. Scientific Journals (ISI)

- **T. Gutierrez**, A. Bergel, C. E. Gonzalez, C. J. Rojas, M. A. Diaz, *Systematic Fuzz Testing Techniques on a Nanosatellite Flight Software for Agile Mission Development*, in IEEE Access, vol. 9, pp. 114008-114021, 2021, doi: 10.1109/ACCESS.2021.3104283. [15]

1.7.2. International Conferences and Workshops

- **T. Gutierrez**, A. Bergel, C. E. Gonzalez, C. J. Rojas, M. A. Diaz, *Toward Applying Fuzz Testing Techniques on the SUCHAI Nanosatellites Flight Software.* Conference paper. Presented in 2020 IEEE Congreso Bienal de Argentina (ARGENCON), 2020. [16]

1.8. Outline of the Thesis

This thesis is organized as follows: Chapter 2 presents fundamental concepts and the work-related regarding CubeSats' fundamental concepts, software design, testing techniques, and fuzz testing in other contexts, then Chapter 3 gives the context of this thesis by describing the SUCHAI flight software. Chapter 4 details the methodology developed to apply fuzz testing to the SUCHAI flight software. The results of this application and the systematic methodology to report and address the failures found are presented in Chapter 5. Chapter 6 lists the threats to the validity of the experiments and analyzes its applicability to other architectures and flight software. Finally, Chapter 7 presents the main conclusions of this work, and Chapter 8 highlights open issues to address in future works.

Chapter 2

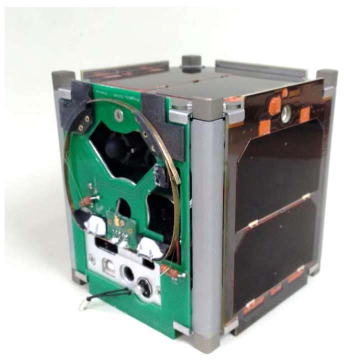
Conceptual Framework

This chapter presents fundamental concepts and a state-of-the-art review of this master thesis to provide context for this work.

2.1. Fundamental Concepts

2.1.1. CubeSats

As mentioned in Chapter 1, CubeSats are a class of nanosatellites. These are built to standard dimensions (Units or "U") of 10 cm x 10 cm x 10 cm. 1U CubeSats typically weigh less than 1.33 kg. They also can be 2U, 3U, or 6U in size. Figure 2.1.a and Figure 2.1.b show the prototypes for the SUCHAI I, and SUCHAI II and III CubeSats, respectively. SUCHAI I is 1U in size, and SUCHAI II and III CubeSats are each one 3U in size.


(a) SUCHAI I CubeSat (1U)

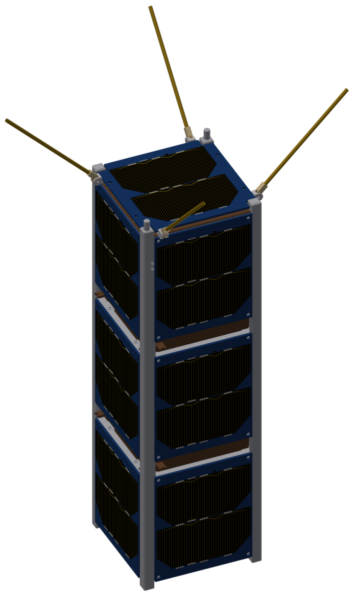

(b) SUCHAI II and III CubeSats (3U)

Figure 2.1: SUCHAI Nanosatellites series prototypes

The original aim of CubeSats' projects was to ensure affordable access to space for university researchers. This low-cost solution was extended to include scientific and educational institutions worldwide, public initiatives in several countries, and eventually private enterprises.

2.1.1.1. System's Communication and Operation

The communication with the satellite in orbit is established using radio links through a ground station, a collection of equipment installed on the earth's surface. This communication consists of sending telecommands from the ground station to the satellite or sending the mission's relevant data from the satellite to the ground station. Figure 2.2 describes this behaviour.

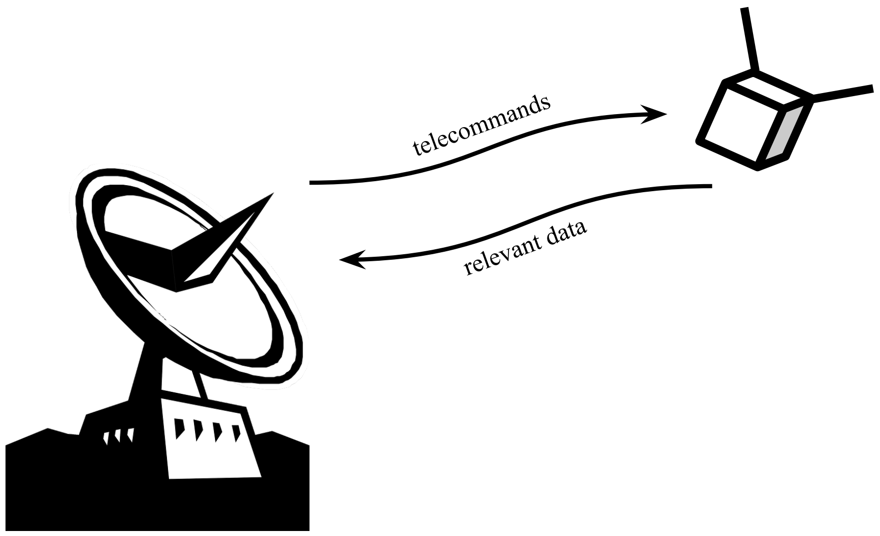


Figure 2.2: Basic system's communication for CubeSats.

The satellite performs its onboard operations autonomously or by receiving telecommands from the ground station and prepares the data to be sent through the command and data handling subsystem (C&DHS). The principal component in a C&DHS architecture is the onboard computer (OBC). An OBC is a unit flying onboard the satellite, which provides processing capability to execute its operations. The application software running on the satellite's main onboard computer is called the flight software.

2.1.2. Testing Techniques in CubeSats' missions

2.1.2.1. Hardware in the Loop Simulation

Hardware in the loop simulation (HILS) is a technique widely used in the development and testing of complex real-time embedded systems in a comprehensive, cost-effective, and repeatable manner [7]. This technique is commonly applied in space systems and, therefore, to nanosatellites' missions. HILS consists of the emulation of sensors, actuators, and mechanical components in a way that connects all the I/O of the electronic control units (ECU) to be tested. Figure 2.3 shows a basic diagram of a HILS system. The simulation monitors the output signals of the system under test and sends generated input into the system under test. The output signals from the system under test usually include actuator commands and operator display information. The output of the embedded system serves as input to the simulation.

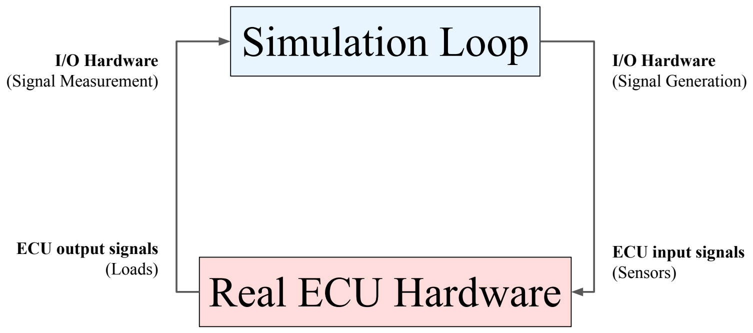


Figure 2.3: Basic HILS System.

2.1.2.2. Software in the Loop Simulation

Software in the loop simulation (SILS) is also a widely used technique for nanosatellites' testing. SILS consists of coupling partially integrated software with an environment simulation for the models of the controllers. Instead of using electrical interfaces for I/O of the electronic units, software interfaces provided by the operating system are used. Since this technique does not require any hardware target, it allows executing tests in early development stages. Figure 2.4 shows a basic diagram of a SILS system.

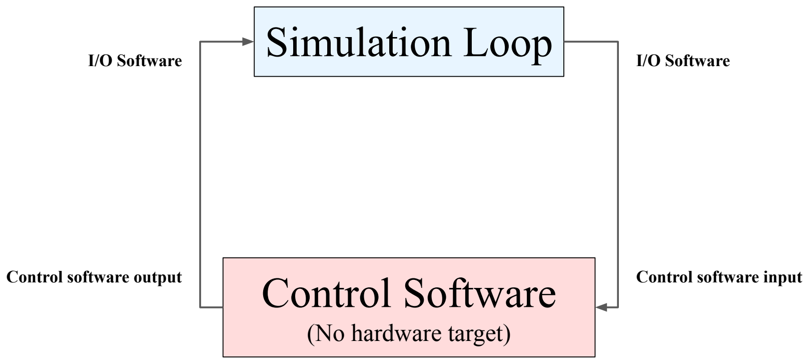


Figure 2.4: Basic SILS System.

2.2. State of the Art

2.2.1. CubeSats' software design

Concepts like software portability and rigorous software design are present in the current related work and have been a topic of discussion because of the recent rise of CubeSat deployments. Coelho *et al.* (2016) [17], Coelho (2017) [18], Ivanov & Bliudze (2020) [19], Gonzalez *et al.* (2016) [20] and Araguz *et al.* (2018) [21] have contributed to this line.

Coelho *et al.* (2016) [17] and Coelho (2017) [18] present the NANOSat MO Framework, which is a standard onboard software framework for nanosatellites implemented in ESA's OPS-SAT mission. This work is based on the CCSDS MO framework and relies on the concept of portability to maximize reuse and customizations between different missions and user needs, with a modular and flexible design. This idea is achieved by turning the onboard software into apps. In this context, an *app* is defined as an onboard software application that can access the peripherals and can be started, monitored, stopped, killed, installed, uninstalled, and updated from the ground. The architecture chosen for the software implementation depends on the number of the running apps, but the swap between architectures is not complex since the interface towards the app developer remains the same. The framework also comes with a software bundle. This work introduces the concept of portable apps in the space field, differing from the cFS contribution in systems' capabilities from the resources point of view.

Ivanov and Bliudze (2020) [19] propose a rigorous and robust way to design software. They present the BIP framework, a component-based language to develop correct-by-construction applications. BIP allows to formally model complex systems and provides a toolset for verification and validation, and code generation. This framework was used in the CubETH CubeSat to design the logic for the satellite's operation and compile it into machine code, which is later executed on the onboard computer. Their approach ensures the overall system's reliability, modularity, and portability. The CubETH mission is based on four main scientific objectives and uses a miniaturized low-power C&DH system and COTS components. Because of the memory limitations of the microcontroller used for the control and data management subsystem, Cortex-M3, the authors had to reduce the model created with BIP. Despite the restrictions, the demonstration of this reduced model on the CubeSat board was considered successful.

Gonzalez *et al.* (2016) [20] propose a hybrid framework to guide software development modeling of nanosatellite missions in an academic environment. The authors highlight that due to the lack of experience that growing countries have in the research and development of satellite technology, there is a shortage of specialized software engineers to work on these types of missions. The proposed model, named Hybrid-Academic-Aerospace Model for Software Development (H4ASD), is based on the ECSS-E-ST-40C[1] documentation and processes, and the disciplines workflow and artifacts of the Rational Unified Process (RUP) to facilitate the assimilation by traditional software engineers with an incipient knowledge in the aerospace field. H4ASD was validated by designing the control and monitoring software of the Libertad-2 3U CubeSat, developed in Universidad Sergio Arboleda, in Colombia. H4ASD uses an iterative and incremental method, following a sequential lifeline, and takes complementary approaches from conventional software engineering concepts and the operating constraints of the space context.

[1] Software engineering standard for space systems' projects. More information can be found at https://ecss.nl/standard/ecss-e-st-40c-software-general-requirements/.

Araguz *et al.* (2018) [21] present three generic design guidelines to improve the system robustness, modularity, and autonomy quality attributes of nanosatellite software architectures. These guidelines were applied to the onboard software architecture for the Cat-1 CubeSat, developed at the Technical University of Catalonia. The authors propose three critical and generic quality attributes to avoid ambiguities as far as possible since qualitatively assessing them is, primarily, a subjective task. The proposed guidelines to improve them consist of encapsulation and goal-oriented decomposition of functionalities, modularization, and the provision of autonomous mission planning capabilities. The application of these recommendations on Cat-1 resulted in a hierarchical ordering of software components, a payload-oriented modularization, and a secure and reliable communication interface that connects low-level modules with the autonomous system.

The core Flight System (cFS) is an open-source flight software solution being developed at NASA. The aim of the project includes reducing time to deploy high-quality flight software, reducing project schedule, and reducing cost uncertainty by facilitating formalized software reuse [22]. The cFS has a solid flight inheritance from NASA projects, and it has also been used in nanosatellites.

Researchers of the Intelligent Space Systems Laboratory (ISSL) at the University of Tokyo have developed the Command-Centric Architecture (C2A), a flight software solution focused on reusability and flexible on-orbit reconfiguration capability [11]. Authors report having used the software on the Hodoyoshi-3 and 4, the PROCYON, and EQUULEUS satellites.

2.2.2. CubeSats' testing techniques

The most common testing techniques for CubeSats found in the literature are directly attached to hardware testing. Kiesbye *et al.* (2019) [5] present and evaluate an environment for HILS and SILS tests with the inclusion of the electrical domain for low-cost satellite development. The tested satellite was MOVE-II, developed at the Technical University of Munich. The results obtained are related to the verification of MOVE-II's attitude determination and control algorithms, the verification of the power budget, and the training of the operator team with realistic simulated failures before launch. Additionally, they present how they used the simulation environment to analyze detected issues after launch and verify the performance of new software developed to address the in-flight anomalies before software deployment. The testing environment described in this work generates results for both hardware and software components of MOVE-II. According to the authors, the environment is potentially suitable for inclusion in a continuous deployment workflow where code changes trigger automatic tests on the hardware. However, they do not report full automation for test cases generation.

Other software testing techniques found in the literature usually imply an exhaustive definition of test cases based on the requirements. Hishmeh *et al.* (2009) [23] show the design, implementation, and testing of the flight software for KySat-1, a picosatellite developed in the Kentucky Space consortium and launched in 2009. The testing methods applied to the software were firmly based on the requirements and documentation. Thanks to applying testing methodologies to the flight software, most bugs were found in the early stages of the development process. This task begins with requirement analysis. After this stage, the flight software team formulated a test strategy and began the test planning. Each bug found was reported after the test cases generation, scripting, and execution. Although the software development team faced problems associated with the time planning of students, they did not propose a new development or testing methodology strategy but a new organization strategy. This issue exemplifies how arduous testing is for small groups developing CubeSats in an academic environment. In software development and testing, time planning and agility are crucial to producing a reliable system, especially in groups with those attributes.

Johl *et al.* (2014) [24] present a reusable C&DH system as part of a series of CubeSat missions being built at Austin Texas Spacecraft Laboratory (TSL), University of Texas. The key idea of this system is to support various system requirements, using a centralized architecture with one main flight computer controlling the actions and the state of the satellite. The authors affirm that flight software testing is an integral step in the development process. Therefore, white-box and black-box testing techniques were planned and applied to validate development. The testing technique applied to the C&DH system was unit testing. The authors propose to apply command execution testing and day-in-the-life testing as future work [24]. Day in-the-life testing refers to verifying the functionality of the fully integrated satellite while a sequence of operations is executed. From this thesis perspective, this type of testing is considered HILS. Also, a graphical user interface for the ground station was developed to minimize the required effort for the ground station operator to interact with the satellite during the testing phase and for flight. They do not mention the methodology to generate the test cases nor an automated testing technique for the software verification.

Schoolcraft *et al.* (2016) [6] present a description and analysis of MarCO mission development. MarCO is a twin CubeSat mission developed by the NASA Jet Propulsion Laboratory (JPL) to accompany the InSight (Interior Exploration using Seismic Investigations, Geodesy and Heat Transport) Mars mission lander. MarCO refined the approach of all the development stages to solve the challenges of quickly building low-budget spacecraft to fly to Mars, relying on components reusability of previous missions. According to the authors, the MarCO flight software development occurred in a tight loop. They focused on a hardware level since computer resources optimization was considered a development requirement. Therefore, the testing techniques applied to the flight software were mainly associated with HILS.

Zaidi *et al.* (2019) [25] present a testing, and a verification and validation (V&V) automated platform to identify anomalies, to characterize their impact, and to reduce costs of system development for CubeSat missions. The platform, part of the Model-Based Systems Engineering (MBSE), bridges the gap between after design and before qualifications phases by first taking information from the concept exploration, definition, and design phases as the input to be processed. Moreover, a software called Missurance controls the test and V&V equipment and receives data when tests are performed. Therefore, the software can notify whether the results meet the functional and design requirements and the test specification. The platform was also used for functional verification and thermal validation of a transmitter. Since the work focused on the interaction of both physical and virtual parts of the system, the mentioned types of testing are mainly HILS and SILS.

cFS and C2A also apply software testing techniques. In the case of cFS, a unit test suit is provided, but the community has produced SILS interfaces using Simulink and the NOS3 spacecraft simulator [22, 26]. For C2A, Nakajima *et al.* have reported the advantages of the command architecture to implement SILS and HILS, and the availability to test the same software with both techniques with minimal source code modification.

2.2.3. Fuzz Testing in Other Contexts

The testing systems applied for the flight software of CubeSats are not profoundly discussed in the literature of this area. In general, the approaches that were found in the related work mention the use of unit testing, HILS and SILS methodologies, or software tools that facilitate the data and command handling from the ground station, but in no case consider automated testing techniques that could be useful for time optimization, which is one of the most common problems for flight software development. This thesis proposes the application of fuzz testing as an automated testing technique that follows the agile development required to perform CubeSat space missions. However, it is possible to find advanced fuzz testing techniques in other areas.

Babić and Bucur *et al.* (2019) [27] propose a system for an automated fuzz driver generation: Fudge. This system operates with an already developed fuzzer, which has found several security and robustness bugs at Google projects. Fudge generates fuzz driver candidates for libraries based on existing client code. A fuzz driver is a test harness, which in this case, exercises the library code. This process accelerates the current fuzz system, enabling fuzz testing more C and C++ codebases. The Fudge high-level overview consists of a backend pipeline where the candidates are generated, and a user interface where developers can track the results. The backend pipeline has three main modules. At first, code snippets are extracted from the library usages. Then, these code snippets are mutated and transformed into fuzz targets. The last module builds and runs the candidate fuzz targets. After the candidates are generated, there is still a manual selection to ensure test consistency. Three different case studies are shown in that work, with the objective to evidence the system's effectiveness. Fudge has found over 150 bugs, which have already been fixed, including eliminating various exploitable security vulnerabilities. The work is an example of what advanced fuzz testing techniques can achieve in other contexts and serves as a guide to lead advanced testing processes for flight software in the nanosatellites' area.

Chapter 3

The SUCHAI Flight Software

SUCHAI CubeSats. SUCHAI is a CubeSat-based space program that includes SUCHAI I, SUCHAI II, SUCHAI III, and PlantSat nanosatellites. Students, engineers, and researchers from different areas develop these nanosatellites in the Space and Planetary Exploration Laboratory (SPEL) of the University of Chile. SUCHAI I is the first CubeSat created in Chile, launched in June 23th, 2017 from the Satish Dhawan Space Centre [28, 29]. The following versions, SUCHAI II, III, and PlantSat, continue developing and updating their functionalities, and they are expected to be launched between 2021 and 2022. These satellites use the SUCHAI flight software[2], a software solution developed for CubeSat nanosatellites designed to be highly modular and extensible. This flight software runs on specific x86, ARM, AVR32, and ESP32 platforms, and it is executed through generic commands. These commands can be executed automatically from specific modules of the software itself, or they can be sent from the ground station as described in Figure 3.1. In previous work, Gonzalez *et al.* (2019) [10] document the design and implementation of the SUCHAI flight software. This section will cover and explain the most relevant parts of their work.

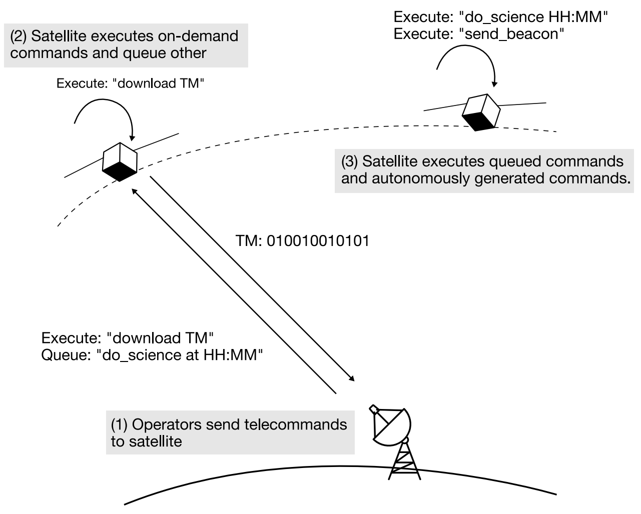

(2) Satellite executes on-demand commands and queue other
Execute: "download TM"
Execute: "do_science HH:MM"
Execute: "send_beacon"
(3) Satellite executes queued commands and autonomously generated commands.
TM: 010010010101
Execute: "download TM"
Queue: "do_science at HH:MM"
(1) Operators send telecommands to satellite

Figure 3.1: Example of satellite operations. Adapted from "An Architecture-Tracking Approach to Evaluate a Modular and Extensible Flight Software for CubeSat Nanosatellites" by C. Gonzalez, C. Rojas, A. Bergel, and M. Diaz, vol 7, pp. 126409-126429, 2019.

[2] https://gitlab.com/spel-uchile/suchai-flight-software

Advantages of the SUCHAI Flight Software Architecture. The SUCHAI flight software architecture is based on the command design pattern adapted for implementation in the C programming language. Figure 3.2 illustrates the application layer architecture. The flight software acts as a generic command executor, and all of its functionalities are encapsulated as commands. The *client* modules request the commands and derive them to the *invoker*. The *invoker* enqueues the commands and makes decisions about their executions to finally send the requests to the receiver. The receiver executes the function associated with the command in the same order they were enqueued [10].

There are two essential advantages of the command pattern architecture. First, the operational requirements are mapped to commands, and commands are mapped to functions. Thus, the software robustness can be examined, and testing commands' execution can track the associated high-level mission requirements. Second, the command execution follows a single path, independently of the software interaction method. Whether the serial console, the communications interface, or a new dedicated *client* is used to interact, the complete command execution mechanism is being tested, which benefits the test coverage. Therefore, different testing techniques can be integrated into the SUCHAI flight software with minimal code instrumentation thanks to the implemented architecture.

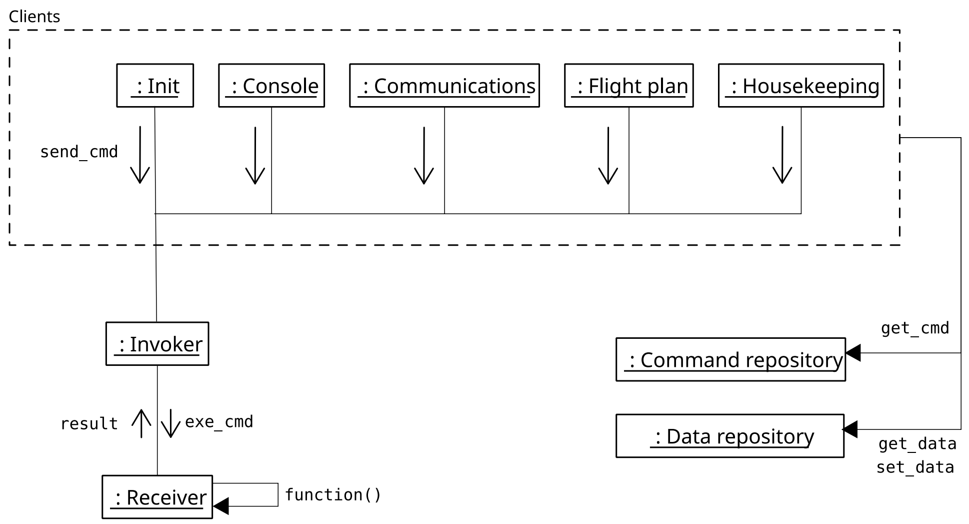


Figure 3.2: The SUCHAI flight software architecture. Adapted from "An Architecture-Tracking Approach to Evaluate a Modular and Extensible Flight Software for CubeSat Nanosatellites" by C. Gonzalez, C. Rojas, A. Bergel, and M. Diaz, vol 7, pp. 126415, 2019.

Building Commands. Commands are grouped in different modules of the SUCHAI flight software. The availability and definition of some of these commands depend on the specific configuration of the SUCHAI flight software. This configuration is set at build time, using a macros-based configuration file. This configuration includes the hardware architecture as well as the operating system.

Each module provides a commands' initialization function that runs when the flight software starts its execution. Listing 1 shows an example of two commands' initialization in a dummy module called "*module1*". Through the function called "*cmd_add*", a command is added to the commands' repository, which is a list that contains the command name, the function it executes, the format of its parameters, and the number of parameters that the command accepts.

```
void cmd_module1_init(void)
{
    cmd_add("module1_cmd1", module1_cmd1_func, "%d", 1);
    cmd_add("module1_cmd2", module1_cmd2_func, "%d %f %s", 2);
}
```

Listing 1: Example of commands' initialization of a specific module of the SUCHAI flight software.

Sending Commands. A command is sent for its execution as a structure named *cmd_type* through a function called *cmd_send*. This structure contains the command id, the parameters, their format and number, and the function that runs the command. To initialize this structure's variables, the structure that matches with the name of the command sent is searched in the commands' repository. If the command's name does not exist, the structure's variables are not initialized.

Commands can be sent by interacting with the flight software through a command-line interface (CLI) or an interface for serial communication. The main OBC is a NanoMind A3200[3] device, which supports FreeRTOS as its real-time operating system kernel. FreeRTOS does not provide a CLI by default. Therefore, the developers designed a serial communication interface to interact directly with the flight software on the OBC while developing and performing testing operations in the laboratory. However, physical communication over the serial port with the OBC is not possible once in orbit. In that case, the flight software is operated through the ground station, which establishes radio link communication using the Cubesat Space Protocol (CSP)[4].

CSP is a small protocol stack designed to run on embedded systems, but not limited to. Currently, LibCSP, the library that implements this protocol, runs on FreeRTOS, Linux, MacOS, and Windows. Specifically on Linux, LibCSP can use ZMQ[5] to communicate different nodes. In particular, the SUCHAI flight software uses this messaging library to connect to other nodes through a Python interface. To pass messages between nodes, a ZMQ Forwarder Device was implemented to collect and forward these messages. This forwarder device must be running in the background before starting the nodes' instances. The communication ports are saved as macros in the configuration file of the SUCHAI flight software. Therefore, the ports can be easily changed by modifying this file. Also, a Python ZMQ CSP node was implemented as an example to test the communication with the SUCHAI flight software. Figure 3.3 represents CSP communication between a SUCHAI flight software instance and the implemented Python test node.

The Python ZMQ CSP node was written as a Python class. This class has attributes, such as the node IP, input and output ports, and the queue containing the messages that will be sent from the test node, among others. Once the node is created and set to run, the method that writes messages and the method that reads them are run as threads. Both threads create and connect to a socket and do read/write operations until the node is stopped. When the node is no longer running, each thread closes its socket's connection.

Current Testing Practices. Unit testing, integration testing, and HILS are the primary testing techniques applied to the SUCHAI flight software during its development to improve

[3] https://gomspace.com/shop/subsystems/command-and-data-handling/nanomind-a3200.aspx
[4] https://github.com/libcsp/libcsp
[5] https://zeromq.org/

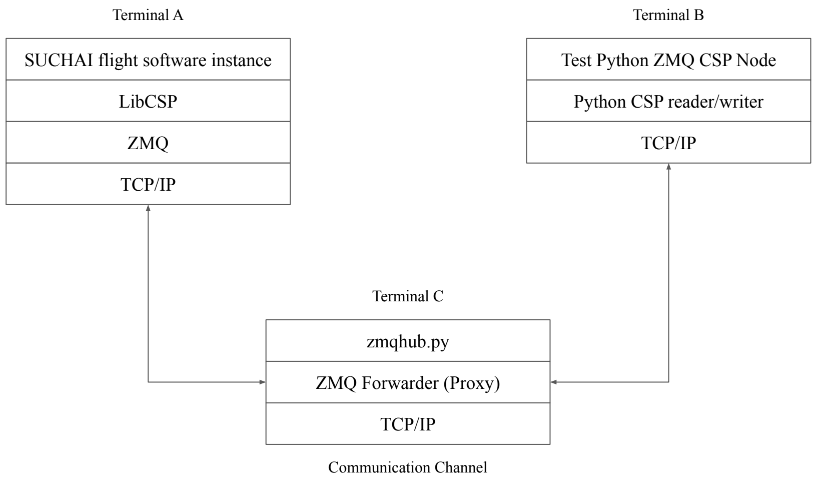


Figure 3.3: CSP Communication between a SUCHAI flight software instance and the Python test CSP node.

and verify particular aspects of its quality. Unit testing was implemented using CUnit. The current unit testing system is based on testing the interfaces of the main modules, but it contains at most four test functions for each module. The integration testing system of the SUCHAI flight software consists of running the flight software with a specific configuration, that is, sending the commands under test with fixed parameters, thus covering only particular use cases. In the case of HILS testing, the software is tested on the same onboard computer that will be installed on the satellite or the satellite flight model itself, which requires a careful test cases' design and environment preparation (software, hardware, and facilities) prior to tests execution in a controlled environment.

The validation methodology of the SUCHAI flight software architecture uses software engineering tools. Specifically, a visual architecture evaluation tool tracks the flight software's quality attributes, generating visualizations that measure the software components' modularity. This tool is complemented with automatic cross-compilation and automated testing to evaluate the software's portability and reliability [10]. In addition, unit testing, integration testing, and visualization generation have been included in a continuous integration system build using the GitLab CI/CD tools.

Chapter 4

Fuzz Testing

Fuzz testing is an automated software testing technique that feeds a random input into a program to uncover system failures. *Software failure* is defined as an unexpected software behavior that gives a different result from the expected one. There are three main types of software failures: loss of service, incorrect service delivery, and system/data corruption [12].

Section 4.1 describes how fuzzing was applied to find unexpected failures on the SUCHAI flight software. The SUCHAI flight software architecture determines the complexity of this application. However, as discussed later, nothing prevents this approach from applying to different flight software. Section 4.2 lists the different employed strategies. Finally, Section 4.3 presents some initial aspects when the experiment was run.

4.1. Fuzz Testing in the SUCHAI Flight Software

As explained in Chapter 1, the SUCHAI flight software is considered a critical embedded system because it carries out the whole system control procedures of the nanosatellite. Therefore, the objective is to find vulnerabilities associated with the system's availability and reliability. These vulnerabilities will be searched in the core of the flight software, which is generic for the different nanosatellite missions. The experiments in the SUCHAI flight software will be run with a specific configuration. The configuration specifications will instruct the flight software to run on an x86-64 Linux distribution.

There are many ways to apply fuzz testing on the SUCHAI flight software, such as sending random input to functions, modules, or commands. The approach of this work uses this technique with commands because it is advantageous in terms of the software architecture. As explained above, the SUCHAI flight software architecture is based on the command design pattern, which means that all of the functionalities are implemented and executed as commands. Thanks to its design, the software provides interfaces to receive commands as inputs through the satellite communication system (that can be emulated in the local loop), the serial console (or Linux terminal), the flight plan, or another specific application task. These interfaces will be used to interact with the SUCHAI flight software running process during the execution of the tests. The result of sending a combination of random commands with a random number of parameters and/or random values of parameters will be analyzed on each test. Thus, each test case should be composed of a sequence of commands.

The implementation of fuzz testing in the SUCHAI flight software uses the fuzzing architecture proposed in *The Fuzzing Book* (Zeller *et al.*, 2019) [14], which provides a Python Runner and Fuzzer classes. The Runner represents the process to be executed with the randomly generated data. Each of the Fuzzer classes represents a system that generates and feeds this data into a consumer. In this context, FSRunner is a class inherited from Runner and interacts with the SUCHAI flight software. FSRunner has methods that run this process with the fuzzed commands and parameters. The Fuzzer classes inherit from a Fuzzer base class and have methods to generate a sequence of random commands and parameters. The method to generate a sequence depends on the strategy to be chosen.

To start communication between the fuzzing application and the SUCHAI flight software, first, a ZMQ Forwarder Device starts its execution from the beginning of the fuzzing application execution until the end. Every time a sequence is randomly generated with the Fuzzer, the Runner invokes the method *run_process*. *run_process* instantiates the SUCHAI flight software as a child process and starts a Python ZMQ CSP node instance from which the sequences will be sent. The SUCHAI flight software runs with a specific configuration, which includes the ports to receive commands from the node and send messages to the node.

Once communication between both applications is set up and established, the fuzzing application iterates over the fuzzed sequence, sending each command and its parameters as a string in a CSP packet. The SUCHAI flight software can receive commands through its communications module, which receives the sent CSP packet and parses it as a *cmd_type* command structure. Then, this structure is sent for their execution using the *cmd_send* function. After a sequence is sent, the current running process of the SUCHAI flight software is terminated by sending an existent specific command designed for that.

The exit code and the memory usage are obtained through the process identifier. The total CPU time is measured from the fuzzing application since the SUCHAI flight software is instantiated until the process ends the execution.

4.2. Strategies

The implementation of fuzz testing for the SUCHAI flight software[6] is based on four strategies defined by the number of commands sent per sequence, the number of parameters sent per command, and the randomness to produce commands or parameters in a sequence. As mentioned in Section 4.1, each strategy is represented by a different Fuzzer class.

Strategy 0: Random commands. Since the SUCHAI flight software provides a check system for wrong names of commands, this strategy's key idea is to prove the robustness of the SUCHAI flight software with random and possibly unknown commands. This proof can be achieved by providing sequences of random names of commands without parameters. Thus, the implemented Fuzzer creates N random names of commands.

The fuzzer class representing this strategy inherits from the Fuzzer base class shown in *The Fuzzing Book* (Zeller *et al.*, 2019). The Fuzzer base class provides the method *fuzz*, which generates a random string, given a minimum and a maximum number of characters in a given range of the ASCII code. In this case, the minimum and maximum lengths were 0 and 10, respectively, and the ASCII characters range from 0 to 127 Unicode codes. Listing 2 shows an example of a sequence of 5 random generated commands represented as a list of commands.

Each randomly-generated command name is added to a list to generate a random sequence. A command is generated by calling the *fuzz* method from the Fuzzer base class. Once a sequence is created, the Runner's *run_process* is invoked to send the sequence to the SUCHAI flight software instance.

[6] https://gitlab.com/tamigr.2293/SUCHAI-FS-Fuzzy-Testing

```
[
  "\u0002.f\u0006\u000bSLyf",
  "\u0005Hw_YP\u0015O\u0007",
  ";\u0013\bR",
  "\n-v\u00157=",
  "l-&05v\u0014<0 "
]
```

Listing 2: Example of a random sequence of 5 commands using Strategy 0.

This strategy should not make the software crash because of the check system mentioned above. Before the *communications* module sends the command object to the *invoker*, it checks if the command name exists in the *command repository.* If the command name does not match with any of the registered command names, it is not directed to the *invoker* to execute.

Strategy 1: Random number of parameters. By providing sequences of known commands with a random number of parameters, this strategy mainly searches for possible errors in the implementations of commands that are not considering the number of the passed parameters. The fuzzer class representing this strategy also inherits from the Fuzzer base class shown in *The Fuzzing Book* (Zeller *et al.*, 2019). In this case and the following ones, the commands' names are not randomly generated but randomly chosen from the list of commands implemented in the up-to-date version of the SUCHAI flight software. The list of commands used has approximately 87 commands.

The generated commands' names and the parameters are added to separated lists. The number of parameters is randomly chosen. This number ranges from 0 to 11, which is the maximum number of parameters a command has. Each parameter is a random value of a fixed type. The implementation of the commands contains the available types' definitions. These are `int`, `long`, `unsigned int`, `float`, and `string`. There are specific methods in the class to generate random values of each type. The random generated values of the types `int`, `long`, `unsigned int` and `float` range from the minimum to the maximum value each type can take in C. The length of each randomly generated value of the type `string` ranges from 0 to 10 characters. Listing 2 shows an example of a sequence of 5 random generated commands represented as a list of commands.

```
[
  "rw_get_speed",
  "gssb_get_temp",
  "com_update_status s +N6X^u 1493284797 QV5] -99074046229686539454141128783806425363 Y@vp]2@ -1445638141
  -9183061798728961283 6099342389522771552 232463368718254291156507652637438239844",
  "tm_parse_status -`&&C 8357191077871478161 +h +Pv:J=Q 6561626627886653267",
  "eps_update_status a>& Kk+Z+ePMY -2080916401"
]
```

Listing 3: Example of a random sequence of 5 commands using Strategy 1.

Strategy 2: Random parameter values with randomly chosen types of values. This strategy provides known commands with the exact number of expected parameters, but the values and types of these parameters are random. The types of the values are randomly chosen, too; therefore, they may not necessarily correspond with the expected types of values. The goal is to mainly find errors in the implementations of commands that may cause a crash because they do not check for the values, the values type, or the variables range.

The fuzzer class representing this strategy inherits from the created class in the previous strategy. The commands are also generated from the list of implemented commands. After the fixed number of the parameters that a chosen command receives is obtained, the random types of the parameters are chosen. Then, the random method to generate each parameter type is called to generate the random values. The parameters are also added to a list. Listing 4 shows an example of a sequence of 5 random generated commands represented as a list of commands.

```
[
  "rw_get_speed",
  "com_send_cmd 2215228249495620775 -1061103018",
  "drp_test_system_vars",
  "com_reset_wdt 8038059285502060246",
  "tm_send_all -524353203349985004 -709326626292023698"
]
```

Listing 4: Example of a random sequence of 5 commands using Strategy 2.

Strategy 3: Random parameter values with defined types of values. This strategy looks for errors in implementations of commands that have unchecked properties of values, such as the length of each parameter. Therefore, the strategy's design consists of sending known commands with the exact number of parameters that each commands receives, where each parameter is a fixed value of a defined type. Unlike the previous strategy, the values' types must correspond with the expected types in this case.

The fuzzer class representing this strategy inherits from the created class in Strategy 1. The commands are also generated from the list of implemented commands. After the fixed number of the parameters and the parameters' types that a chosen command receives are obtained, the parameters' random values are chosen. Then, the method to generate each parameter type is called to generate the random values. The parameters are also added to a list as in the previous strategies. Listing 5 shows an example of a sequence of 5 random generated commands represented as a list of commands.

```
[
  "gssb_msp_cal_temp -1056747876 -2078397161",
  "com_update_status",
  "com_send_rpt -857193128 W",
  "drp_set_deployed 1656682019",
  "gssb_msp_get_temp"
]
```

Listing 5: Example of a random sequence of 5 commands using Strategy 3.

4.3. Execution

The different strategies were executed by sending sequences containing 5, 10, 50, and 100 commands. Each of these sequences with a predefined size was generated 1,610 times for each strategy to find helpful test cases. Therefore, there were 25,760 sequences executed on the SUCHAI flight software in total. Initially, the execution of the 25,760 sequences lasted around three days. In an experiment replication with the same sequences, the execution lasted 175,872 seconds. Then, the replication lasted two days and 53 minutes. The experiment was replicated to analyze time execution on a different computer system with more processing

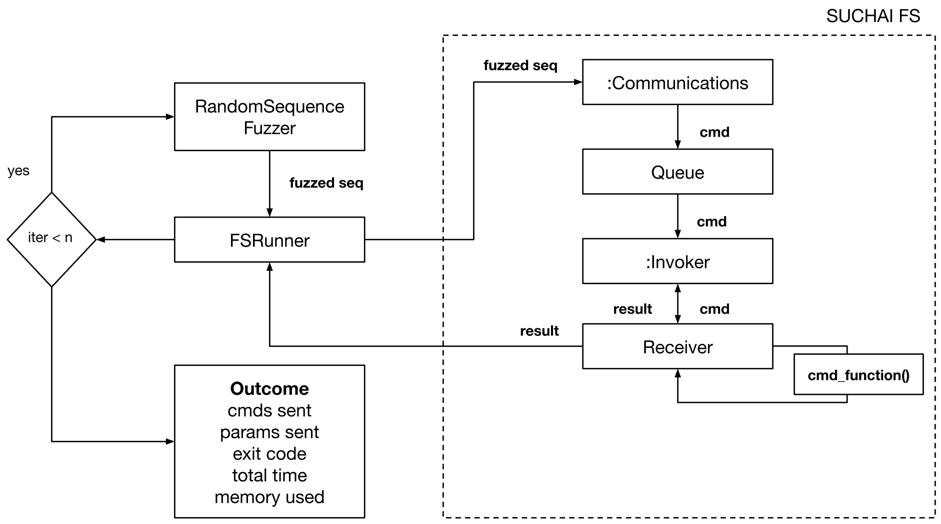


Figure 4.1: Logic diagram of the proposed fuzz testing implementation and the communication system with the SUCHAI flight software. RandomSequenceFuzzer is the system to generate the random sequence of commands to be sent to FsRunner. FsRunner interacts with the SUCHAI flight software running process. It sends the sequence commands to the running process. The SUCHAI flight software receives the commands through the communications module and executes them following the logic of its architecture.

and storage capacity.

Chapter 5

Results

The results obtained from the execution of the strategies mentioned in Section 4.2 are analyzed in terms of the exit code, execution time, and memory consumption for every sequence. For each strategy, 6,440 sequences were executed. These sequences were equally distributed in four sets based on the contained number of commands: 5, 10, 50, and 100 commands per sequence.

5.1. Experiment execution results

Initially, the experiment was executed under the operating system Ubuntu version 18.04. The hardware used was an Intel(R) Core(TM) i5-6200U processor @2.3 GHz and 12 gigabytes of RAM.

For strategy 0, the results show that the failure rate by sending random names of commands without parameters is 0 %. Then, the results are consistent with the hypothesis that the software validates the names of the commands before they are sent for execution.

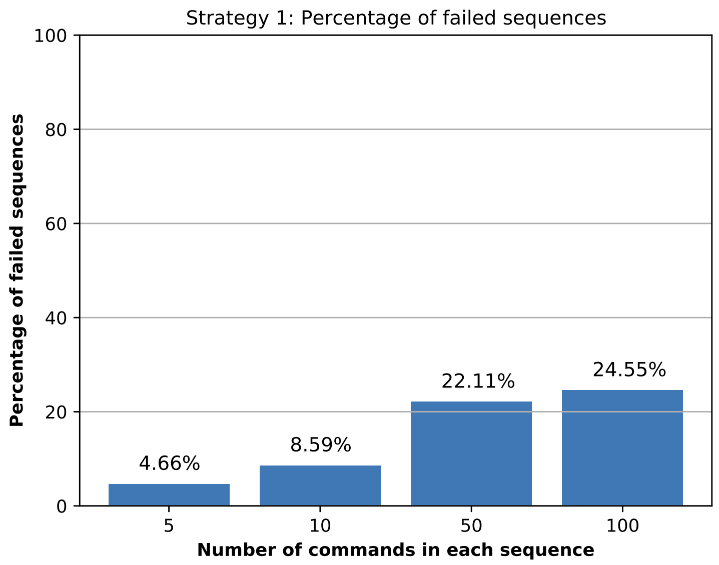


Figure 5.1: Percentage of failed sequences of commands given a fixed number of commands per sequence for strategy 1.

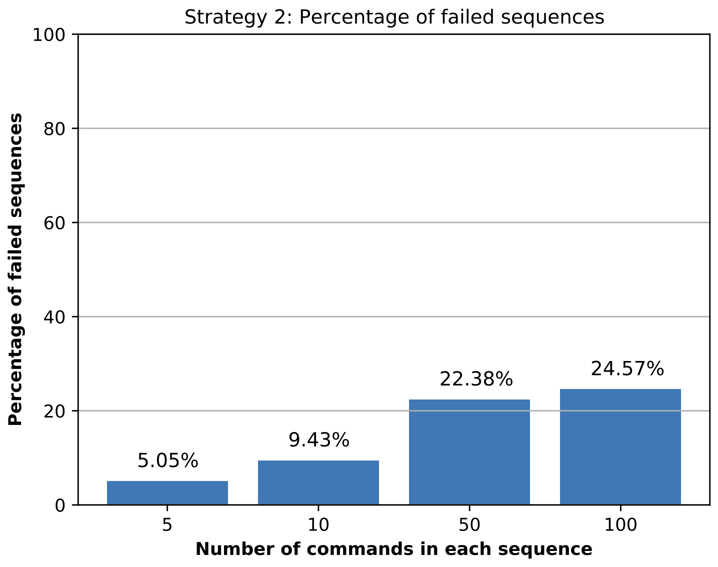


Figure 5.2: Percentage of failed sequences of commands given a fixed number of commands per sequence for strategy 2.

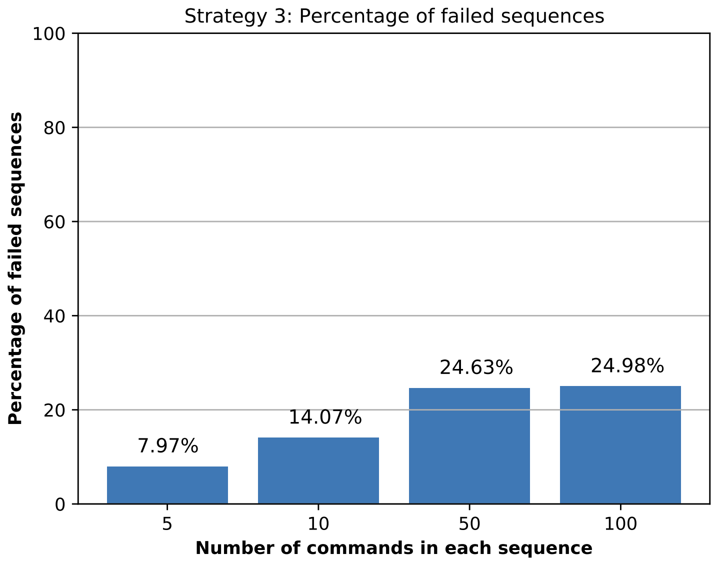


Figure 5.3: Percentage of failed sequences of commands given a fixed number of commands per sequence for strategy 3.

The percentages of the failed sequences on each set for strategies 1, 2, and 3 are shown as bar charts in Figure 5.1, Figure 5.2, and Figure 5.3, respectively. The variable in the x-axis is the number of commands per sequence. The variable in the y-axis is the percentage of failed sequences compared to the total number of sent sequences per strategy.

For each of the above figures, there is an increase in the failure percentage between the sets. Since the random generation of commands and parameters uses a uniform distribution, the probabilities of choosing parameters that make a command execution crash the SUCHAI

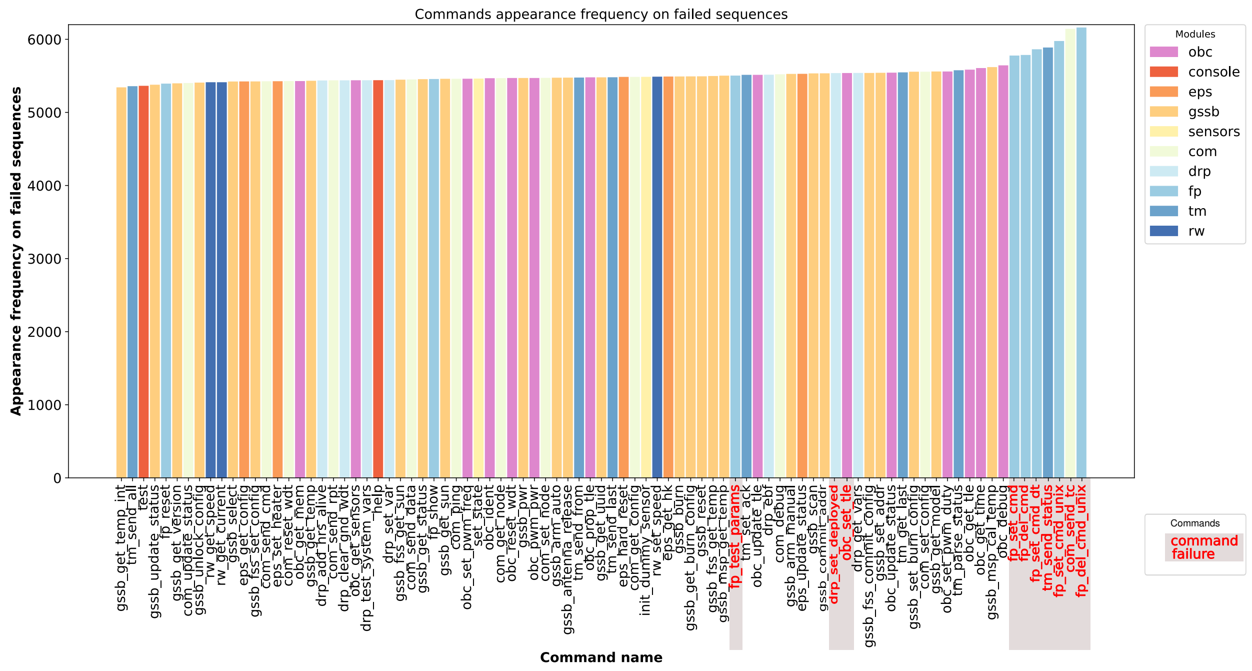


Figure 5.4: Commands appearance frequency on the failed sequences, classified by command type (module). The commands that made the SUCHAI flight software crash are red-colored in the x-axis labels (identified command failure).

flight software process increases as the number of commands contained in a sequence is greater.

The maximum time that a sequence took to execute was approximately 6,463 seconds. Ten sequences lasted longer than 200 seconds to execute, which makes up only 0.15 % of the sequences. This behavior only appeared in the first execution of the experiment; therefore, it is not mainly related to the experiment performance itself but other factors discussed in Section 5.2.

The memory consumption of the sequences varies from 10,268 to 11,100 kilobytes. There is not a significant variation between strategies. In all cases, the maximum memory consumption of a sequence is in the order of 10,000 kilobytes.

Figure 5.4 shows the commands' occurrence frequency on sequences that made the SUCHAI flight software crash, classified by module. Each color represents a module. The red color on a command name of the x-axis labels indicates an identified failure in the SUCHAI flight software produced by the command. The number of times a command appeared in the same sequence was not considered in the counting for more precise analysis. In total, ten commands were identified as a cause of a SUCHAI flight software crashing. Seven of them appeared more frequently in the sequences that made the SUCHAI flight software fail. The module that has the majority of the ten identified commands is the *flight plan (fp)*.

By looking at Figure 5.4 one can identify the commands that made the SUCHAI flight software crash. The developers identified the first seven commands (from right to left) that appear more frequently in the sequences as a cause of failure in the SUCHAI flight software at least once. This identification process will be explained more in detail on Section 5.3.

These results show that, by applying a fuzz testing technique to the SUCHAI flight software, new failures were found. As we mentioned on Chapter 3, unit testing, integration testing and HILS are being applied to the SUCHAI flight software as traditional testing methods. However none of these techniques reported these specific failures previously.

There have been found sequences that made the SUCHAI flight software crash. From these sequences, ten failing commands were particularly identified by the software development team. Also, anomalies in the execution time of the sequences were found, which will be analyzed and discussed on Section 5.2.

5.2. Experiment replications results

As mentioned at the beginning of this chapter, three experiment replications were performed to measure the execution time under other conditions with better hardware resources. The hardware used was an Intel(R) Core(TM) i7-990X @3.47GHz, and 24 gigabytes of RAM. The objective of replicating the experiment on different hardware is to verify whether the findings mentioned in Section 5.1 relate or not to the employed hardware. In terms of software, these replications were performed under the operating system Ubuntu version 20.04. The results related to memory consumption and exit code were also measured again to be consistent.

In contrast to the first execution of the experiment, there are no significant differences in the execution time of the sequences. None of the sequences lasted longer than 200 seconds to execute. The differences between the experiments are associated with the help of better resources to replicate the experiment. However, more experiments are necessary to associate a definite cause to this effect and achieve more confidence about the obtained results to make statistical conclusions. This threat is discussed in detail in Section 6.1.

Strategy 0 does not present any sequence with execution time longer than 10 seconds, which is the expected behavior since a random command name should not be recognized as valid input in the first place. This kind of inputs does not cover any more code than the necessary statements to validate them.

As in the first experiment execution, there is not a significant variation of the memory consumption between the sequences. The variation ranges from 11,655 to 12,279 kilobytes.

There are differences between the original experiment execution and its replications in the number of failures, with more failures in the first experiment execution. In addition, some values of the memory consumption measurements are equivalent to 0 kilobytes in the experiment replication. These findings could be associated with the conditions under which the replications were executed and will be further discussed in Section 6.1.

Differences in execution time, number of failures, and wrong values in the measurements of memory consumption are not related to an experiment performance issue. These differences will be further discussed on Section 6.1.

5.3. Failures Fixing and Characterization

Once the sequences were sent to the SUCHAI flight software and the relevant results from their execution were identified, the findings were reported to the software development team. In eight sprint sessions, the authors identified the bugs, fixed them, and characterized them for a detailed analysis. This process shows that a systematic fuzz testing technique can be applied and integrated into the SUCHAI flight software, not only as an automated testing application, but also as an agile methodology to test this flight software.

Sprints. At the beginning of each sprint session, the reports made for the software development team were analyzed. This analysis consisted of searching for the sequences that made the SUCHAI flight software crash and reproducing them manually, sending the specific commands that make up each sequence to the SUCHAI flight software, one by one. In parallel, each software team member tried to identify a failed sequence. When a member found a sequence, the issue was reported in Git, the version control system used to track the SUCHAI flight software code changes. The information attached to the issue report was the number of commands in the sequence, the exit code returned by the execution of the sequence on the SUCHAI flight software, and the commands of the sequence with their respective values of parameters[7].

The changes made in the code to fix the issues found were attached to the bug reports on the version control system. This process made it possible to keep track of the error type, architecture level affected, modules affected, the number of code lines changed, and the number of modified functions.

Once the issue associated with a failed sequence was identified and fixed, three questions were asked to the software team members better to understand the failure and the complexity of its solution. The possible answers to these questions are represented as a number scale from 1 to 5, ranging from “very unimportant/very easy” to “very important/very difficult”. The questions are the following:

- How important is the failure?
- How difficult is the failure to find?
- How difficult is the failure to fix?

ID	Command name	Exit Code	Error type	Where is it being executed?			Criticality	Ease of finding	Ease of fixing	Architecture level			Affected modules	#LOC*		#Funcs.**
				SAT	GND	SIM				ORG	EXP	FIX		+	-	
#4	fp_del_cmd_unix	-6	SS	🟦	🟦	🟦	4	3	4	D	A	D	data_storage.c data_storage.h cmdFP.c	256	119	10
#5	tm_send_status	-6	FA	🟦		🟦	5	2	3	A	A	A	repoData.c cmdCOM.c cmdCOM.h cmdTM.c taskCommunications.c	72	36	3
#6	obc_set_tle	-11	SF	🟦		🟦	4	3	1	A	A	A	cmdOBC.c	1	1	1
#7	drp_set_deployed	-11	NP	🟦		🟦	4	2	1	A	A	A	cmdDRP.c	5	8	1
#8	com_send_tc	-6	SS		🟦	🟦	3	5	5	A	A	A	cmdCOM.c	1	1	1
#9	fp_del_cmd	-11	NP	🟦		🟦	5	2	1	A	A	A	cmdFP.c	16	18	1
#10	fp_del_cmd_unix	-11	NP	🟦	🟦	🟦	4	1	1	A	A	A	cmdFP.c	9	11	1
#11	fp_set_cmd_dt	-6	SS	🟦		🟦	4	3	3	D	A	D	data_storage.c globals.h	4	3	1
#12	fp_test_params	-11	SF				1	2	1	A	A	A	cmdFP.c	5	7	1
#13	fp_set_cmd_unix	-11	SF	🟦		🟦	4	2	1	A	A	A	cmdFP.c	10	11	1
#14	fp_set_cmd_dt	-11	SF	🟦		🟦	4	2	1	A	A	A	cmdFP.c	10	12	1
#15	fp_set_cmd	-11	SF	🟦		🟦	4	1	1	A	A	A	cmdFP.c	18	20	1

(*) # of code lines to fix de bugs
(**) # of modified functions to fix the bug

Table 5.1: Characterization of the failures found in the SUCHAI flight software.

Results. In total, 12 failed sequences were identified and fixed by the developers during the sprint sessions. Each of these sequences failed because of the crashing on the execution of one

[7] https://github.com/spel-uchile/SUCHAI-Flight-Software/issues?q=is:issue+label:Fuzz-Testing

particular command. Ten commands had identified errors. From the questions asked during the sprint sessions and the tracking of the code changes, the failures were characterized, as shown in Table 5.1. This description includes the ID of the issues reported in the version control system and the command directly associated with the failure. The exit code refers to the values of the POSIX signals sent to the process to terminate its execution. The error type is the main part of the error message associated with the process exit code. Errors are reported in the table with particular acronyms: *SS* is a *stack smashing*, *SF* is a *segmentation fault*, *NP* is a *null pointer* and *FA* is a *failed assertion.* "*Where is it being executed?*", "*Criticality*", "*Ease of finding*", "*Ease of fixing*", and "*Architecture level*" attributes were part of the discussion with the software development team during the sprint sessions. Therefore these answers represent the developers' opinion from 1 to 5, where 1 means that the bug being studied is irrelevant for the mission/not difficult to find/not difficult to fix, and 5 means that it is critical for the mission/very difficult to find/very difficult to fix. In the table, the acronyms shown below the previously mentioned question represent the places where a certain command is executed: *SAT* is the onboard satellite, *GND* is the ground station, and *SIM* is the simulator. The architecture level from where the failure originates (*ORG*), expressed (*EXP*) and fixed (*FIX*) could be the drivers layer (*D*) or the application layer (*A*). The affected modules, number of added (+) or extracted (-) code lines to fix the bug and the number of modified functions to fix the bug were extracted from the version control system after the bug was fixed.

As discussed previously, the majority of the software failures found are related to the *flight plan* module. The *flight plan* module contains almost all of the error types, except one: a failed assertion. Besides, four of the eight commands associated with the flight plan are executed onboard the satellite. *"fp_del_cmd_unix"* is executed on the ground station and the simulator. *"fp_test_params"* is just a testing command; therefore, it is not executed in any of the shown modules. It is important to note that *"fp_set_cmd_dt"* and *"fp_del_cmd_unix"* appear twice on the table because there were different failures found on each of these commands.

The criticality is strongly associated with the platform where the command is being executed. The developers considered that eight out of the ten presented commands are critical since they are being executed onboard the satellite, while *"com_send_tc"* was rated as 3 in criticality level because it is executed only on the ground station and the simulator. *"fp_test_params"* was rated as 1 since it is not executed in any of the mentioned parts.

Fixing the issues. The bug related to the command *"com_send_tc"* is considered the most difficult to find. The developers tried to identify the cause of failure only by using the debugger but also through trial-and-error, making direct changes to the code until the software did not crash anymore. The rest of the bugs were rated in the range from 1 to 3 regarding the "*Ease of finding*" category. The developers found eight out of twelve bugs by sending commands with no parameters. The first bug associated with the command *"fp_del_cmd_unix"* was rated as 3 because it was necessary to find the precise configuration of the database system to reproduce it. The failure related to the command *"tm_send_status"* is a failed assertion independent of the values of each parameter, though it is relatively easy to find. The first bug associated with the command *"fp_set_cmd_dt"* is a stack smashing type of failure, where a buffer saves a string without its null character. Though the bug is not difficult to find, the developers required time to understand the cause of the failure.

Four out of twelve bugs were rated with a value higher than 1 ("very easy") on the attribute *ease of fixing*. Eight bugs were rated as 1 because of a wrong parameter validation when sending commands with no parameters, which are considered easy to fix. The developers considered that the bug related to the command *"com_send_tc"* was the most difficult to fix since, as mentioned above, the process to fix it was not direct. A missing implementation of the functionalities of a particular database system caused the first bug associated with the command *"fp_del_cmd_unix"*. Thus, the complexity for fixing this bug lies in the number of functionalities, and therefore code lines, that must be implemented to execute this command correctly under the required configuration for that database system. According to the developers, the bug associated with the command *"tm_send_status"*, and the first bug related to the command *"fp_set_cmd_dt"* are not very hard to find, but a certain level of knowledge is required to solve them.

The first bug associated with the command *"fp_del_cmd_unix"* has the most significant numbers of modified lines of code and modified functions to fix the bug, which are 375 and 10, respectively. These variables affect its complexity, which was mentioned by the developers beforehand. The bug associated with the command *"tm_send_status"* has 108 modified code lines and three modified functions. The rest of the bugs do not present a value higher than 20 and 1 on the attributes *# of code lines to fix the bug* and *# of modified functions to fix the bug*, respectively.

This section of the characterization shows that the majority of the failures were easy to find and easy to fix by the developers. This indicates that this technique could not necessarily find complex failures. However, it did find several failures in around three days of execution. Furthermore, a systematic identification, registration, characterization and fixing of these bugs were carried out in eight sprint sessions.

Impact on the architecture. All of the bugs were expressed in the application layer of the software architecture. Ten out of twelve bugs were originated from and were fixed on the application layer. Only two bugs were originated from and were fixed on the drivers' layer. Both are considered critical and are related to the flight plan module. The driver to interact with the different database systems is implemented on the `data_storage.c` file. Since the last-mentioned bugs are originated from the drivers' layer, `data_storage.c` is an affected module.

5.4. Fuzzing Into a Continuous Integration System

As we mentioned on Chapter 3, the remote version of the SUCHAI Flight Software source code is being built and tested under Gitlab continuous integration tools. The visualizations' generation, the compilation and the testing process carried out to validate and track the remote repository source code changes are defined as separate stages in the YML configuration file of the CI system. All of these stages run in a Docker container under a virtual server of Amazon Web Services (AWS).

The testing stage includes unit testing and integration testing, where each of these techniques is defined as a job[8]. Recently, fuzzing was implemented in the continuous integration system after evaluating the results obtained from the local experiment executions[9]. This

[8] https://docs.gitlab.com/ee/ci/jobs/

[9] Sofia Bobadilla collaborated in this section. More documentation about this work can be found at https://gitlab.com/spel-uchile/suchai-flight-software/-/blob/feature/framework/docs/fuzz_testing.md. Also, there is a presentation with the obtained results at https://docs.google.com/presentation/d/1U5azfMSIjnOsTTHBJNcF_EQzBoK1gMmRuWfqwiGjmfQ/edit#slide=id.gebe91ce37f_0_280.

technique was included in the testing stage as an additional job. Listing 6 shows the code fragment that contains the definition of this job in the YML configuration file. The job runs a shell script named *install_test_fuzzy-framework.sh*, which contains all the command instructions required to run the test.

```
test_fuzzing:
stage: test_fuzz
script:
   - docker run -v ~/.ssh:/root/.ssh -i suchai-fs bash install_test_fuzzy-framework.sh
   - docker system prune -f
rules:
   - if: '$CI_COMMIT_BRANCH == "feature/framework"'
```

Listing 6: Fuzz testing stage definition in the YML configuration file of the SUCHAI flight software continuous integration system.

Listing 7 shows the *install_test_fuzzy-framework.sh* file. Everytime the test is run on the C.I. system, the SUCHAI flight software is updated and compiled in the Docker container. Once the current version is built, the strategy to run the fuzz test is randomly assigned. Depending on the selected strategy, the test will be run with a certain number of sequences and commands per sequences. If the SUCHAI flight software crashes due to one or more sequences' executions, the script will exit with a failure status. This will make the job fail, notifying the respective status on the pipeline execution.

As shown in Listing 7, the number of sequences and commands to run per strategy are specified as arguments of the *run_experiment.py* file. These two parameters, named *iterations* and *commands_number* respectively, were determined mainly by the execution time, the objective and characteristics of each strategy. The execution time in the CI system was limited to run up to two minutes by developers' consensus. Strategy 3 generates test cases more similar to a real operation of the flight software without mistakes regarding the number of the parameters or their types. Therefore, to simulate a reality's closer behaviour, there shall be more commands' executions in a single instance, that is, more commands in a sequence compared to the number of commands of the other strategies. Table 5.2 shows the current configuration for each strategy. Every configuration was run 30 times to have more precise measurements of the execution time, calculating the minimum and maximum execution time of all iterations per strategy.

Strategy	Executions	No. of sequences per iteration	No. of commands per sequence	Iterations' min. exec. time [s]	Iterations' max. exec. time [s]
1	30	15	12	96.06	96.11
2	30	15	12	93.05	93.67
3	30	15	18	98.1	99.12

Table 5.2: Current test configuration (number of sequences and commands) for each strategy and execution time in the continuous integration system.

Only the *run_experiment.py* file was modified to set up the execution in the new environment, get the execution status and retrieve clear information about the generated sequences and the results. To set up the execution in the new environment, the corresponding execution path and the name of the executable file were reassigned. The exit status from the *install_test_fuzzy-framework.sh* script is captured from the *run_experiment.py* file execution. Then, the file *run_experiment.py* was also modified to throw an exit code that indicates a failure if at least one sequence execution makes the flight software crash, or an exit code

```bash
 #!/bin/bash
cd SUCHAI-Flight-Software
git pull
git checkout feature/framework
version=$(cmake -version)
echo $version

cmake -B build -DAPP=simple -DSCH_FP_ENABLED=0 -DSCH_HK_ENABLED=0 -DSCH_CSP_BUFFERS=2000000
cmake --build build
cd ../SUCHAI-FS-Fuzzy-Testing
strategy=$(($RANDOM%3 +1))
echo "STRATEGY: " $strategy

if [ $strategy = 1 ]
then
   python3 run_experiment.py --iterations 15 --commands_number 12 --strategy 1
elif [ $strategy = 2 ]
then
   python3 run_experiment.py --iterations 15 --commands_number 12 --strategy 2
else
   python3 run_experiment.py --iterations 15 --commands_number 18  --strategy 3
fi

status=$?
cd -
echo "The exit status was $status"
exit $status
```

Listing 7: install_test_fuzzy-framework.sh script.

that indicates success otherwise. Finally, the relevant information, such as the sent sequences and the execution result were printed in the log files to provide quick access to this data.

To date, there has been two software updates in the control version system since the job implementation was finished. Therefore, the job has been executed less than ten times. From these executions, there has not been crashes found. However, in the future, with more executions, the effectiveness of this technique can be analyzed and improved on the continuous integration system. Also, this could help current and future developers to build and spread better development and testing practices.

Chapter 6

Discussion

6.1. Threats to validity

The threats' analysis to the validity of this work is based on the description made by Cook *et al.* (2019) [30].

Conclusion validity. The experiment, defined as sending specific sequences that were initially randomly generated, was reproduced three more times in order to capture more accurate results mainly associated with time and memory consumption. These replications were executed under different conditions that were as similar as possible to the first execution. However, the experiment has low statistical validity because of the few executions and the different hardware and software conditions. A higher number of executions is required to mitigate this threat. Also, the conditions of the experiment reproductions must be defined beforehand. Furthermore, this experiment has random heterogeneity since 1,610 random sequences for each predefined number of commands per sequence were sent on a particular strategy.

Internal validity. There were found very few variations when replicating the experiment. External elements could have affected the executions, possibly attached to the operating system and hardware. These results were expected in the case of memory consumption and execution time. However, the number of failures also varied: fewer failures were found on each replication than in the first execution.

Construct validity. The SUCHAI flight software has a configuration module, which has several variables to configure the execution of the software conditions, such as tasks to be reproduced, database system to be set up, communication system settings, among others. The experiments were executed under only one standard configuration, considering that only the software was tested. Combinations of values for configuration module variables were not tested. However, several strategies were developed to analyze different types of scenarios. Besides, the results considered the number of failures, memory consumption, and execution time.

External validity. The implementation for this work applies only to the SUCHAI flight software context. However, it is possible to generalize it to flight software with similar software architecture, although changes to the source code might be necessary. The experiment was performed in a specific version of the software to help the developers implement an improved version of it. After the developers fixed the bugs found with this technique, the experiment was rerun to find new failures, and no new bugs were found.

6.2. Applicability to Other Architectures

As mentioned earlier, the SUCHAI flight software runs on different architectures. The main objective of this work is to find vulnerabilities of the SUCHAI flight software in an x86 Linux platform. However, some of the limitations of applying fuzz testing only in this platform are the drivers and the commands not being tested. Since the SUCHAI nanosatellites' main OBC runs FreeRTOS in a NanoMind A3200 computer, these drivers differ from those tested. Also, specific commands on each satellite are not available in the x86 platform.

This section will cover and discuss the application of the same fuzz testing techniques in the SUCHAI II, SUCHAI III, and PlantSat nanosatellites' main OBC as a complementary part of this work. The objective is to find critical software vulnerabilities that cause a software crash not found in an x86 Linux platform and evaluate fundamental hardware performance variables. This work is performed in the laboratory, where the HILS tests are carried out as part of the satellites' last development stages.

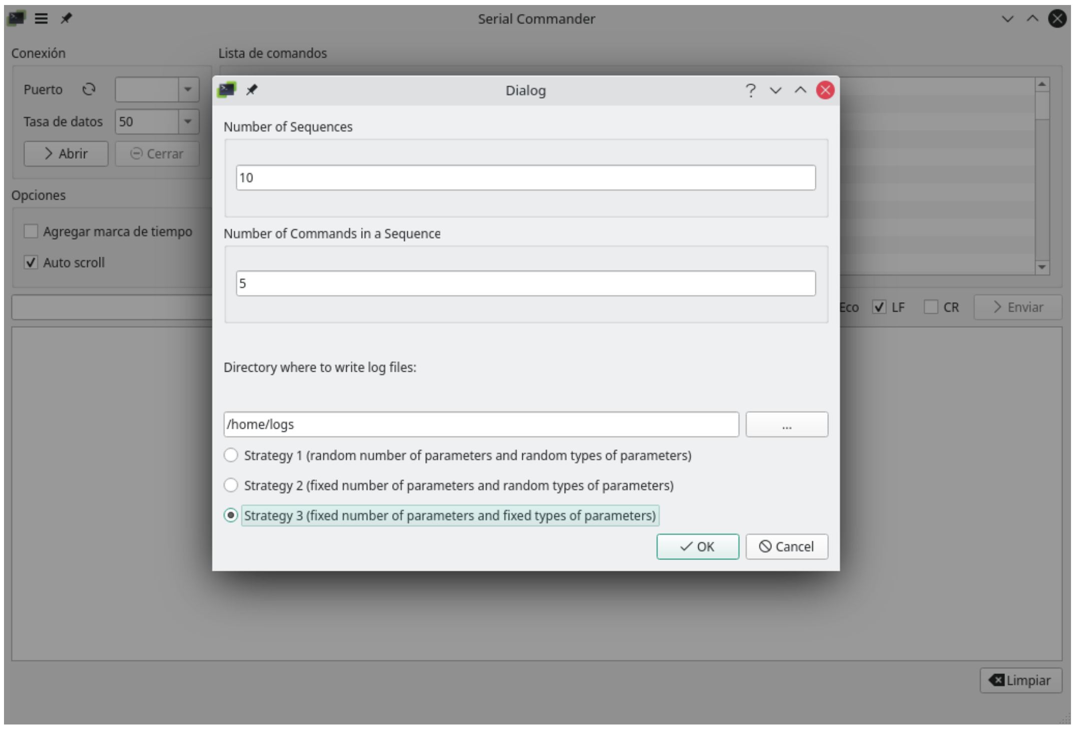
The main idea is to send randomized sequences of commands to the SUCHAI flight software version running on the OBC. As we explained on Chapter 3, there are two methods to interact with OBC. One of these methods is operating the nanosatellite through the ground station using radio link communication. The other method consists of sending the commands through a serial communication interface. Sending commands through a serial communication interface is a direct way to transmit and receive data because it does not require intervening or filtering the sent or received messages.

The developers designed the serial communication interface. The main project has a VCS repository in Gitlab [10]. This interface provides tools for setting the connection port, connecting, sending data directly to the SUCHAI flight software, and saving log files. It was possible to modify this interface to add a tool that automatically generates random sequences and sends them to the SUCHAI flight software.

This new extension [11] includes a menu option to generate the random sequences. When selected, the option will open a configuration window, which asks for the number of sequences that will be generated, the number of commands contained in each sequence, the directory where to save the log files, and the strategy to be used. Figure 6.1 shows this window with an input example, where ten sequences of five commands each will be generated with Strategy 3. After these variables are configured and confirmed, the application automatically generates the random sequences and sends them to the SUCHAI flight software running process over the serial port. The application uses the fuzzer classes of strategies 1, 2, and 3 to generate the random sequences. Strategy 0 is not applied in this case since the generation of random command names will not cause a system's crash. While fuzzing, the application will be processing the execution log files to detect crashes in the SUCHAI flight software. After the fuzz testing is run, the interface will show a report with the name of the log files containing more or less than one SUCHAI flight software restart message, indicating that a failure occurred.

The experiment preparation consisted of evaluating the commands to be chosen and setting up the connection with the nanosatellite. Filtering commands is necessary due to commands' execution that could cause possible irreversible hardware damages. Furthermore, SUCHAI II, III, and PlantSat are in the late stages of development, where the nanosatellites are partially

[10] https://gitlab.com/carlgonz/SerialCommander
[11] https://gitlab.com/tamigr.2293/serial-commander-fuzzing



Puerto

Tasa de datos: 50

[ ] Agregar marca de tiempo

[✓] Auto scroll

[✓] LF [ ] CR

Number of Sequences: 10

Number of Commands in a Sequence: 5

Directory where to write log files: /home/logs

○ Strategy 1 (random number of parameters and random types of parameters)

○ Strategy 2 (fixed number of parameters and random types of parameters)

[✓ Strategy 3 (fixed number of parameters and fixed types of parameters)]

Figure 6.1: SerialCommander serial interface window to set up generating random sequences and sending them automatically.

or fully integrated. A laptop disconnected from the power source and a USB to a serial connector is required to set up the connection over the serial port, as shown in Figure 6.2.

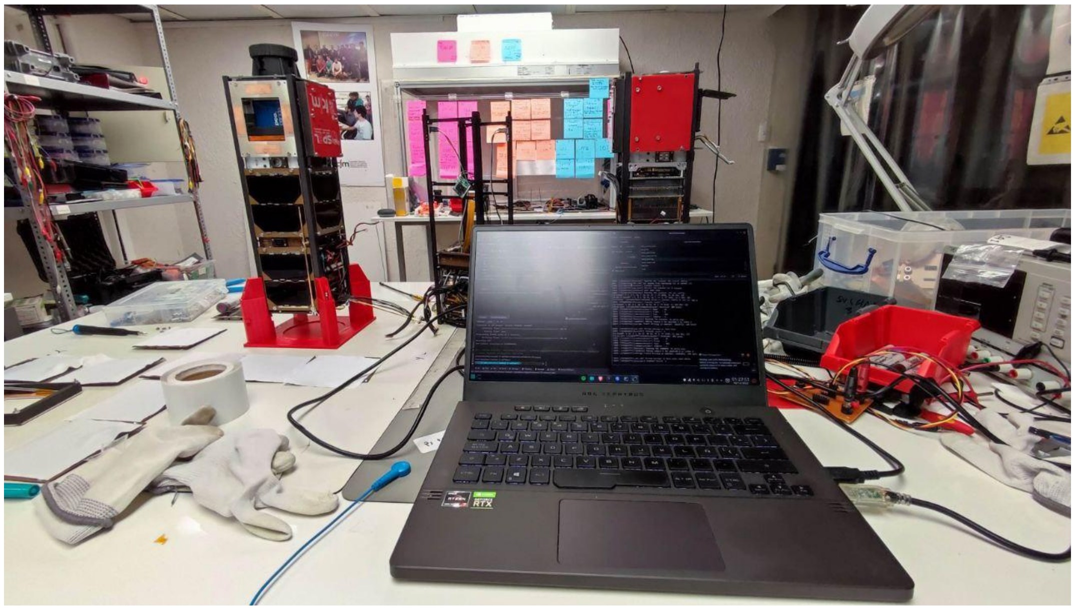


Figure 6.2: Laptop set up to run fuzz testing in a SUCHAI nanosatellite's main OBC through a serial communication interface.

6.2.1. Experiments

First, some preliminary experiments were run to improve the implementation and test the interface into the hardware environment. The first part of the preliminary experiments was carried out in a prototype of a star tracker payload outside the laboratory. The second part of the preliminary experiments was run in the OBC of the SUCHAI III nanosatellite. The final experiments were run in the OBC of both PlantSat and SUCHAI III nanosatellites. SUCHAI II was not tested due to its limited availability during the final development stage. However, it is worth mentioning that each nanosatellite mission has a different but very similar configuration of their main bus.

6.2.1.1. Star Tracker

The SPE Lab Open Star Tracker (SOST) [31] is a payload designed for attitude determination in the SUCHAI missions. This payload also runs the SUCHAI flight software, and it communicates with the main OBC through commands. The tested prototype runs the SUCHAI flight software over a Raspberry Pi Zero W. This prototype was decoupled from the system. Besides, the Raspberry Pi Camera was not included in the hardware system while executing the experiments since it was not available.

The main idea of this first part of the preliminary experiments is to send random sequences to the star tracker computer by connecting it, through the serial port, to the SerialCommander serial interface.

Set Up. The SUCHAI flight software is running as a *systemd* service on the Raspberry Pi. Originally, it was set to restart if the process exited with a non-zero exit code, was terminated by a signal, an operation timed out, or when the configured watchdog timeout was triggered. However, the process must be restarted even if exited with a zero exit code since the implementation sends a command to restart the process right after a randomized sequence is sent. The flight software can not restart by itself because the command to restart it calls the function exit() with a zero exit status, which in the Raspberry Pi OS only exits the process. Therefore, it is required to restart the service to send the next sequences to the process. This was achieved by setting the *systemd* service *restart* option to restart the process on both cases of a failed and succeeded process exit.

Raspberry Pi OS provides a configuration option to enable the serial port through the raspi-config configuration tool. Enabling the option is required to execute the experiments through the serial interface. However, in the SOST, this option was not originally enabled. Therefore, this variable must be modified by accessing the raspi-config configuration tool.

To enable that *systemd* service reads/writes from/to the serial console, the standard input and standard output of the *systemd* service must be set to the device file of the serial port. Originally the device file was not set. Listing 8 shows the configured systemd service.

Execution and Results. Three preliminary experiments were set to run in the SOST with 10, 60, and 500 sequences per strategy. Each sequence contained three randomized commands for the first preliminary experiment. For the second and third preliminary experiments, each sequence contained five randomized commands. The final experiment in the SOST was set to run with 500 sequences of commands per strategy. Each sequence contained five randomized commands.

The results of the experiments indicate that the execution of seven sequences on the SUCHAI flight software caused a failure. These results are still being processed to generate

```
[Unit]
Description=Start SUCHAI Flight Software after boot
After=network.target

[Service]
ExecStart=/home/pi/suchai-software-template/build/payload-software
User=pi
TTYPath=/dev/serial0
StandardInput=tty
StandardOutput=tty
Restart=always

[Install]
WantedBy=multi-user.target
```

Listing 8: Configured systemd service to run the SUCHAI flight software process that restarts after exit. The standard input and output are set to send and receive data through the serial port.

valid conclusions. Further steps include a systematic process to identify, fix and characterize the failures, as well as done in the main work.

6.2.1.2. OBC

As explained above, the onboard computers of the SUCHAI II, SUCHAI III, and PlantSat nanosatellites runs the SUCHAI flight software in the FreeRTOS operating system in a NanoMind A3200 device. These experiments consist of sending randomized sequences to the main OBC's flight software through the serial port using the SerialCommander interface.

Set Up. Compared to the experiments executed in the SOST, in this case, the running process of the SUCHAI flight software does not require any configuration beforehand. The serial port communication serial port is already enabled, and the process restarts every time the command that does this is run.

Results. Two preliminary experiments on the OBC of the SUCHAI III nanosatellite were executed to improve the interface implementation. After running these tests, three experiments on PlantSat and four experiments on SUCHAI III's OBC were executed. In total, 158 sequences were set to run in the nanosatellites' OBC.

The results indicate that nine sequences made the SUCHAI flight software crash. These results are also being processed to generate valid conclusions and take further steps to provide a systematic technique that can help find and solve critical failures in the flight software running in the OBC of the SUCHAI nanosatellites' missions.

6.3. Applicability to Other Missions

Fuzz testing covers many strategies, including black-box, white-box, or grey-box testing methods. Specifically, fuzzing was implemented as a black-box testing method in this work. Then, from experience gained by implementing it for the SUCHAI flight software, the author derives and describes the essential characteristics of a flight software architecture that may facilitate the application of the black-box fuzzing strategies:

- **Interoperability:** The system should have a clear and well-defined interface to interact with the fuzz testing application.

- **Understandability:** The software architecture should be easy to understand and have a clear structure in order to know how to manage the fuzz testing application.
- **Testability:** The requirements of the mission should be consistent and testable. There must be documentation of the public API in order to apply black-box testing. Besides, the system should have the capacity to capture the test results.
- **Performability:** The system should be fast enough to perform each action in a reasonable amount of time, taking into account how many inputs will be sent.

From the reviewed software architectures by Gonzalez *et al.* (2019) [10], the core Flight System [22] and the Command Centric Architecture [11] present a well-documented architecture that fulfills the characteristics previously highlighted. This clear documentation makes it possible to define a straightforward way to fuzz the flight software as a black-box testing method.

The core Flight System (cFS) is an open-source flight software solution developed by NASA [22]. This software exhibits a layered architecture that hides the hardware and OS specifics while providing a core and application layer with general and mission-specific services. The cFS provides an interface to integrate a new application using a publish-and-subscribe architectural style with a software message bus, allowing interoperability. Thanks to its clear software architecture, it would be possible to create a new fuzz testing application to interact with the rest of the system using the software bus. Messages have a well-defined format (CCSDS), so the supported messages list and parameters can be randomized by the fuzzer. All of the applications are connected to the software bus so the fuzzer can interact with the system by sending request messages and observing response messages. Figure 6.3 explains this proposal.

The Command Centric Architecture (C2A) is the flight software developed by ISSL researchers at the University of Tokyo, and it focuses on reconfiguration capability. A significant feature of C2A is to describe the behavior of the spacecraft by commands and to present a clear software architecture to register and execute both single and block commands [11]. Following the C2A concepts, it would be possible to develop a fuzz testing essential function to send commands and randomize parameters and the execution order as described in Figure 6.4. The block commands concept in C2A matches with the idea of command sequences. The new essential function requires a definition table that aggregates all other existing command definition tables. By fuzzing application-specific block commands in the C2A, it would be possible to explore the effect of uncertainty in executing the individual commands sequences or testing the spacecraft robustness to deviations in the expected operations.

6.3.1. F Prime

F Prime is a free and open-source flight software framework tailored to small-scale systems, developed at JPL, NASA [32]. Thanks to the availability of its source code and documentation, it was possible to create an approach of the application of fuzz testing techniques in order to find sequences of commands that crashed the flight software execution.

The approach for the study of the application of fuzzing into FPrime[12] was implemented and run in a local computer, using the same hardware resources as described in Section 5.1.

[12] https://gitlab.com/tamigr.2293/fprime-fuzz-testing

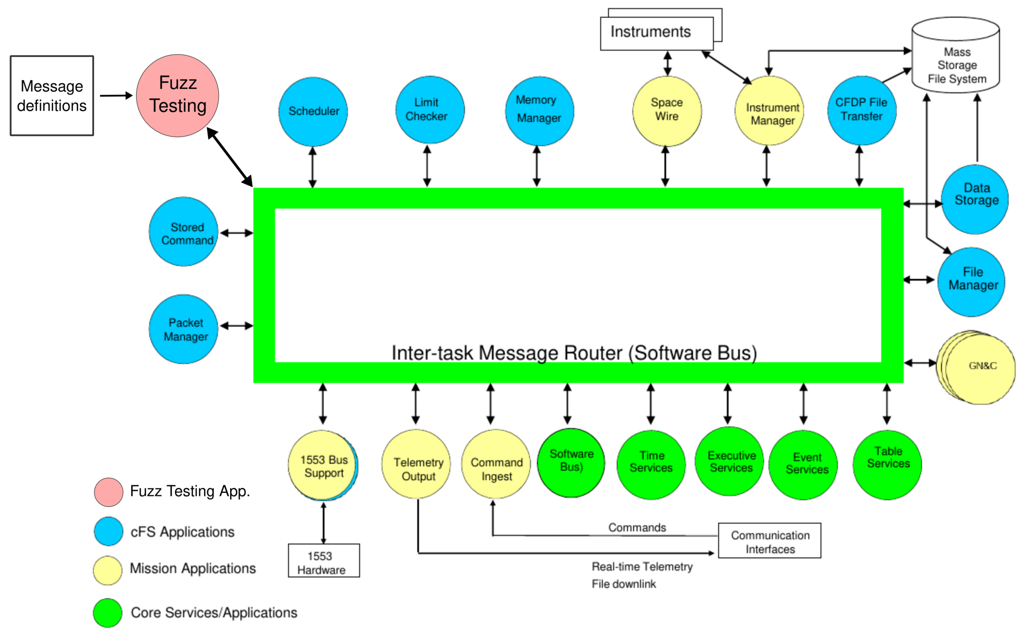


Figure 6.3: cFS top level architecture modified to integrate a fuzz testing application. Adapted from "core Flight System (cFS) Background and Overview", NASA, 2014, https://cfs.gsfc.nasa.gov/cFS-OviewBGSlideDeck-ExportControl-Final.pdf (accessed 2021 June 23)

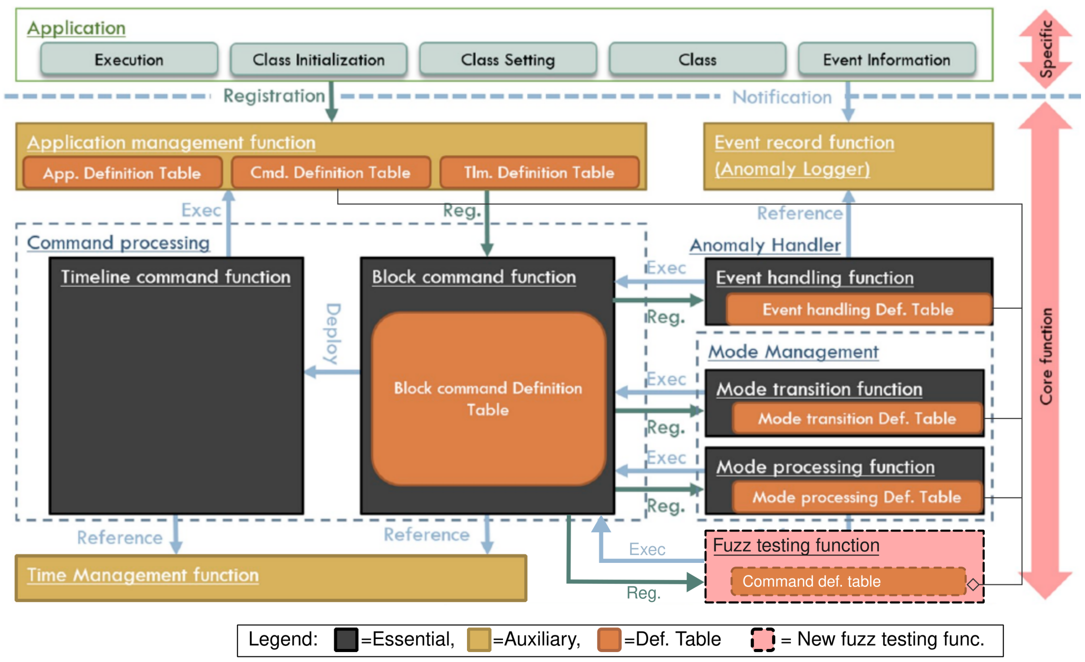


Figure 6.4: C2A software architecture modified to integrate a fuzz testing essential function. Adapted from "Command-centric architecture (C2A): Satellite software architecture with a flexible reconfiguration capability", by Nakajima *et al.*, Acta Astronautica, vol 171, pp. 208-214, 2020.

The implementation uses the F Prime Ground Data System (GDS)[13], from which testers can automate integration tests against F Prime software. In this case, the software and GDS application were run. Commands can be sent to the F Prime software from the GDS using, optionally, its user interface (UI). From the perspective of this approach, the UI is advantageous since it provides a clear interaction with the software. The GDS allows one to send commands, and it retrieves information about their execution as an event description. Events are described by Bocchino *et al.* (2018) as a report of the flight software behavior, and they can be classified into seven categories based on their severity. The objective was to specifically find sequences that produced events with a fatal severity, which is a critical failure event that typically results in an embedded system restart.

The fuzz testing implementation for the FPrime flight software consists of a child class of a Fuzzer and a child class of a Runner, called RandomSequenceFuzzer and FprimeGDSRunner. RandomSequenceFuzzer and FprimeGDSRunner represent a random sequence of commands and the process to be executed, respectively. The generation and the sending of this random sequence to the F Prime software require the GDS GUI interaction. The tool selected to handle this interaction was Selenium WebDriver[14].

As shown in Figure 6.5, RandomSequenceFuzzer access to the GDS GUI, using the *selenium.webdriver* module, to get the necessary information to generate fuzzed command sequences. That is, the relevant information to generate fuzzed sequences corresponds to the available commands' names of the application, the number of parameters of each command, and their expected types. Once the fuzzed sequences are generated, they are sent to the FPrimeGDSRunner. FPrimeGDSRunner sends each fuzzed sequence to the GDS GUI, interacting with the UI elements in the web browser to click on the dropdown lists, write on the field and click the send button to dispatch the command with their parameters. The GDS GUI then sends each sequence command to the embedded application built on FPrime. The application events contain the results of the commands' execution. All the events that occurred in the application execution are sent to the GDS GUI. FPrimeGDSRunner filters the events, only saving those with high severity. The outcome also saves the descriptions of these events.

A command can be sent from the GDS UI to the F Prime software by selecting a command name from a dropdown list and writing or selecting its parameters' values on a field provided for each one. The values of some parameters must be selected from a dropdown list, and the rest must be written in a text field. Each character of the written values is randomly generated given a specific range in the ASCII code. Also, the length of each value is randomly chosen given a particular range. For example, in Figure 6.6 the command eventLogger.SET_ID_FILTER is about to be sent. This command expects two parameters. The first parameter must be written in a text field, for which the fuzzer generates random characters. The second parameter must be selected from a dropdown list. The possible values for this parameter can be obtained from the browser interaction through Selenium. An index of the list is randomly chosen. This index is the position of the value that will be selected from the list. Once the commands and their parameters are selected and written, the command can be dispatched by pressing the "Send Command" button.

The "Events" section tab from the GDS GUI (Figure 6.7) shows the resulting events. This section tab contains a table with all the occurred events. Each event has a name, an ID, a description, a severity classification, and the time it happened. As mentioned, the filtered information only contains the description and classification of high severity events.

[13] https://nasa.github.io/fprime/UsersGuide/gds/gds-introduction.html
[14] https://www.selenium.dev/documentation/webdriver/

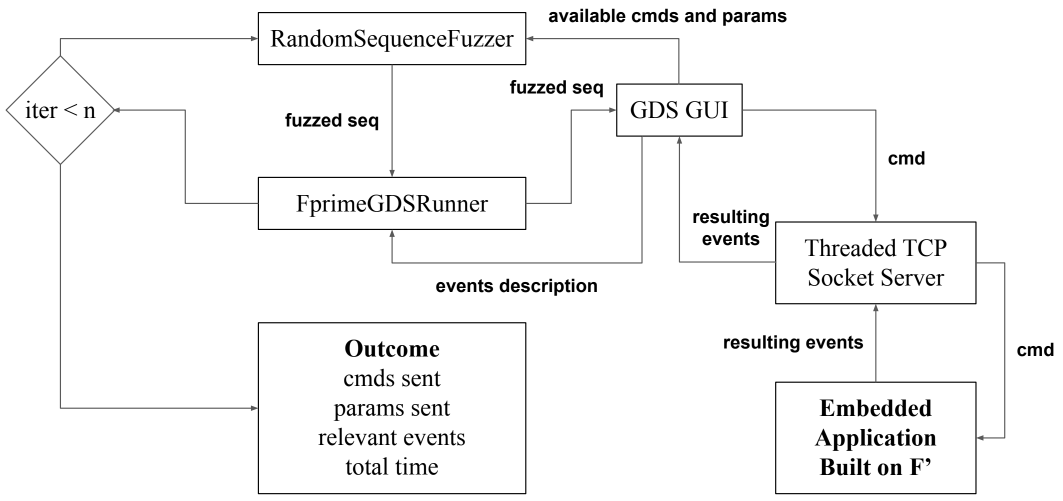
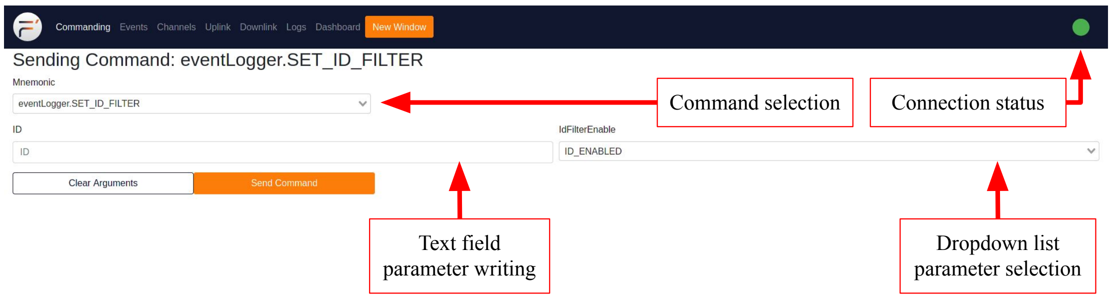


Figure 6.5: Logic diagram of the first approach of a fuzz testing implementation for the FPrime flight software.



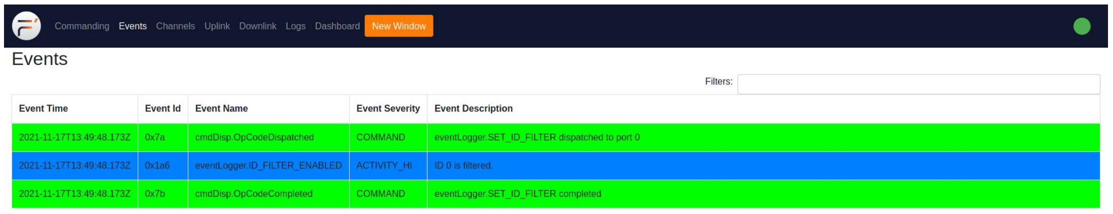
Sending Command: eventLogger.SET_ID_FILTER

Mnemonic: eventLogger.SET_ID_FILTER

ID: ID

IdFilterEnable: ID_ENABLED

Figure 6.6: Commanding section of the FPrime GDS GUI.



Events

Event Time	Event Id	Event Name	Event Severity	Event Description
2021-11-17T13:49:48.173Z	0x7a	cmdDisp.OpCodeDispatched	COMMAND	eventLogger.SET_ID_FILTER dispatched to port 0
2021-11-17T13:49:48.173Z	0x1a6	eventLogger.ID_FILTER_ENABLED	ACTIVITY_HI	ID 0 is filtered.
2021-11-17T13:49:48.173Z	0x7b	cmdDisp.OpCodeCompleted	COMMAND	eventLogger.SET_ID_FILTER completed

Figure 6.7: Events section of the FPrime GDS GUI.

6.3.1.1. Preliminary Experiments

Minimal test cases were executed before the experiments' execution to study the fuzzing application behavior. One of the problems encountered was an interruption in the fuzzing program execution caused by an exception. The GDS application threw this exception due to an interaction problem with the GDS UI using the Selenium framework. Then, this error is not attributable to any of the FPrime applications but the fuzzing application design. Another problem encountered is associated with the commands run in the FPrime software. At first, all the commands available in the FPrime application were executable in the fuzzing program. However, one of them intentionally caused the interruption with the GDS GUI connection. The command that caused this failure was removed from the list of available commands in the fuzzing application. Also, some of the malformed generated input, according to their expected type or expected size, caused exceptions at the GDS UI level. In this case, the GDS UI does not let the affected commands be sent for their execution to the flight software. Then, the behavior of this kind of input in the FPrime software is uncertain under this first approach.

The first three experiments consisted of the execution of 1250 commands, each one. The findings exhibited one fatal event apparently raised by the execution of one command. Only the sequence that contains this command was reproduced to replicate the behavior. However, no failures were found by reproducing the sequence. Then, all the sequences involved in the experiment were reproduced. The event occurred in some replication with this methodology, but not all. The author looked for the execution's registries to better understand why the event did not always occur. It was found out that, by default, logfiles of each execution on the FPrime application are internally saved. These log files showed that the events did not constantly occur in the same order. The commands' execution order changes because FPrime commands may not be dispatched nor executed in the order they were sent. Then, it is not clear whether the execution order of the events affected this result.

Chapter 7

Conclusions

This work reviews the flight software testing strategies used in several CubeSat projects, which indicates that unit testing, SILS, and HILS are the most common techniques. However, not all flight software frameworks or CubeSat missions document the testing procedures to ensure software quality and robustness. Moreover, in search of agile testing solutions, there were not found any reported use of more advanced software testing techniques, such as fuzz testing, to CubeSat missions. Fuzz testing techniques have demonstrated in other areas their usefulness by providing automation to the testing procedures, improving software robustness. For this reason, these techniques' application is proposed in a context of agile and low-cost CubeSat development, which, to the best of the author's knowledge, has not been introduced before.

This work explored the usage of fuzz testing techniques in the flight software of the SUCHAI series of nanosatellites by running a set of strategies. It was found out that the command-based architecture of the SUCHAI flight software facilitates the interaction with the fuzzer. Moreover, testing through commands facilitates the use of these strategies both in early development stages (development machines or continuous integration systems) and qualification/formal functional testing campaigns (protoflight or flight models).

The test results showed that 42.8 % of the total sequences failed during the execution of the tests, which is a sign of active software bugs not found with previous testing techniques (unit testing, integration test, and HILS). After three days of executing more than 1,000,000 commands in an unattended manner, twelve bugs were found in total. These results were appropriately reported to the SUCHAI software team, and these twelve bugs were fixed through eight sprint sessions, identifying their relevant characteristics. In this regard, not only failures in the SUCHAI flight software were found, but also a systematic methodology to report, fix and characterize these failures was proposed. Furthermore, this thesis provides cues for the applicability to other nanosatellite missions from the software architecture point of view and analyzes possible applications in cFS and c2A flight software. Also, it presents and discusses preliminary results of a first fuzzing approach for FPrime flight software as a practical application.

The results of this work show that fuzzing strategies helped to find new failures in the SUCHAI flight software that other techniques did not report before through a systematic process. This systematic process can be integrated into the testing stage by identifying, registering, fixing, and characterizing the failures. Moreover, by modifying only the main script, it could be possible to integrate this application into the Gitlab Continuous Integration System of the SUCHAI flight software VCS repository.

The developers characterized the failures after every sprint session through a questionnaire. The main items that helped determine some advantages and disadvantages of the application are the criticality, the ease of finding the failures, and the ease of fixing them. In particular, the results show that the failures found will not necessarily be highly complex or difficult to find. However, this automated technique helped developers find critical failures in at least five modules of the flight software and gave a quick solution to many failures not solved until that point.

As nanosatellites must be reliable to prevent mission failure, testing applications for these systems deserve to be thoroughly studied and disseminated. Accordingly, this work may help current and future small and nanosatellite missions to improve their quality and thus, reducing the mission risk. The automation possibilities and the unattended execution are crucial to achieving the repetition and agility required to test particular nanosatellites, but also hundreds to thousands of satellites within the context of mega-constellations.

Chapter 8

Future Work

The main part of this work consisted of fuzzing the SUCHAI flight software as a separate task from other testing techniques. However, fuzzing was also executed in later stages of the development when HILS tests were being carried out. That is, the fuzz tests were executed on the SUCHAI protoflight models and showed sequences that made the SUCHAI flight software crash while running on the main OBC of the SUCHAI III and PlantSat nanosatellites. The following steps in this research include, in the first place, processing the results and applying a systematic technique to identify and solve the failures found. In the second place, studying more advanced strategies could significantly increase the number of failures found. For example, more complex strategies could be developed by generating input based on genetic algorithms techniques. The identification of failure paths discovered by code coverage or other dynamic analysis metrics could lead the input generation.

Though the fuzz testing technique was integrated into the continuous integration system of the SUCHAI flight software repository on Gitlab, no failures have been found since the job containing this application has been executed less than ten times. More executions are required to generate valid conclusions and improve the applied technique.

A more extensive analysis based on the execution of more experiments in the fuzzing of FPrime software is required to generate accurate conclusions of the software behavior. As previously discussed, some of the problems were encountered at the interface level of the application. Another possible approach to avoid these issues can be developed by directly executing commands from the FPrime flight software application. The FPrime GDS also provides several tools to facilitate specific tasks, such as the GDS Integration Test API[15] or the support of a sequence format to execute commands in sequence[16], which can be helpful to enhance the first approach or to create other fuzzing applications. Once an application produces accurate results and finds runtime failures, there must be a systematic process to report and fix the failures found.

[15] https://nasa.github.io/fprime/UsersGuide/dev/testAPI/user_guide.html
[16] https://nasa.github.io/fprime/UsersGuide/gds/seqgen.html

Bibliography

[1] T. Villela, C. A. Costa, A. M. Brandão, F. T. Bueno, and R. Leonardi, "Towards the thousandth cubesat: A statistical overview," *International Journal of Aerospace Engineering*, vol. 2019, 2019.

[2] E. National Academies of Sciences, Medicine, *et al.*, *Achieving science with CubeSats: Thinking inside the box.* National Academies Press, 2016.

[3] D. L. Dvorak, "NASA Study on Flight Software Complexity," in *AIAA Infotech@Aerospace Conference and AIAA Unmanned...Unlimited Conference*, (Reston, Virigina), p. 264pp, American Institute of Aeronautics and Astronautics, apr 2009.

[4] J. Finnigan, "A scripting framework for automated flight sw testing: Van allen probes lessons learned," in *2014 IEEE Aerospace Conference*, pp. 1–10, 2014.

[5] J. Kiesbye, D. Messmann, M. Preisinger, G. Reina, D. Nagy, F. Schummer, M. Mostad, T. Kale, and M. Langer, "Hardware-In-The-Loop and Software-In-The-Loop Testing of the MOVE-II CubeSat," *Aerospace*, vol. 6, p. 130, Dec. 2019.

[6] J. Schoolcraft, A. T. Klesh, and T. Werne, "MarCO: Interplanetary Mission Development On a CubeSat Scale," in *SpaceOps 2016 Conference*, SpaceOps Conferences, American Institute of Aeronautics and Astronautics, May 2016.

[7] J. A. Ledin, "Hardware-in-the-loop simulation," *Embedded Systems Programming*, vol. 12, pp. 42–62, 1999.

[8] S. Jeong, Y. Kwak, and W. J. Lee, "Software-in-the-loop simulation for early-stage testing of autosar software component," in *2016 Eighth International Conference on Ubiquitous and Future Networks (ICUFN)*, pp. 59–63, IEEE, 2016.

[9] D. José Franzim Miranda, M. Ferreira, F. Kucinskis, and D. McComas, "A Comparative Survey on Flight Software Frameworks for 'New Space' Nanosatellite Missions," *Journal of Aerospace Technology and Management*, p. e4619, Oct. 2019.

[10] C. E. Gonzalez, C. J. Rojas, A. Bergel, and M. A. Diaz, "An Architecture-Tracking Approach to Evaluate a Modular and Extensible Flight Software for CubeSat Nanosatellites," *IEEE Access*, vol. 7, pp. 126409–126429, 2019.

[11] S. Nakajima, J. Takisawa, S. Ikari, M. Tomooka, Y. Aoyanagi, R. Funase, and S. Nakasuka, "Command-centric architecture (c2a): Satellite software architecture with a flexible reconfiguration capability," *Acta Astronautica*, vol. 171, pp. 208–214, 2020.

[12] I. Sommerville, *Software engineering.* No. P-322, 2011.

[13] P. Godefroid, "Fuzzing: hack, art, and science," *Communications of the ACM*, vol. 63, pp. 70–76, Jan. 2020.

[14] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "The fuzzing book," in *The Fuzzing Book*, Saarland University, 2019. Retrieved 2019-09-09 16:42:54+02:00.

[15] T. Gutierrez, A. Bergel, C. E. Gonzalez, C. J. Rojas, and M. A. Diaz, "Systematic fuzz testing techniques on a nanosatellite flight software for agile mission development," *IEEE Access*, 2021.

[16] T. Gutierrez, A. Bergel, C. E. Gonzalez, C. J. Rojas, and M. A. Diaz, "Toward applying fuzz testing techniques on the suchai nanosatellites flight software," in *2020 IEEE Congreso Bienal de Argentina (ARGENCON)*, pp. 1–4, IEEE, 2020.

[17] C. Coelho, O. Koudelka, and M. Merri, "Nanosat mo framework: achieving on-board software portability," in *14th International Conference on Space Operations*, p. 2624, 2016.

[18] C. Coelho, *A software framework for nanosatellites based on ccsds mission operations services with reference implementation for esa's ops-sat mission.* PhD thesis, Ph. D. dissertation, 2017.

[19] A. B. Ivanov and S. Bliudze, "Robust software development for university-built satellites," *arXiv preprint arXiv:2010.02208*, 2020.

[20] F. A. D. González, P. R. P. Cabrera, and C. M. H. Calderón, "Design of a nanosatellite ground monitoring and control software–a case study," *Journal of Aerospace Technology and Management*, vol. 8, no. 2, pp. 211–231, 2016.

[21] C. Araguz, M. Marí, E. Bou-Balust, E. Alarcon, and D. Selva, "Design guidelines for general-purpose payload-oriented nanosatellite software architectures," *Journal of Aerospace Information Systems*, vol. 15, no. 3, pp. 107–119, 2018.

[22] D. McComas, J. Wilmot, and A. Cudmore, "The Core Flight System (cFS) Community: Providing Low Cost Solutions for Small Spacecraft," in *AIAA/USU Conference on Small Satellites*, aug 2016.

[23] S. F. Hishmeh, T. J. Doering, and J. E. Lumpp, "Design of flight software for the KySat CubeSat bus," in *2009 IEEE Aerospace conference*, pp. 1–15, Mar. 2009. ISSN: 1095-323X.

[24] S. Johl, E. Glenn Lightsey, S. M. Horton, and G. R. Anandayuvaraj, "A reusable command and data handling system for university cubesat missions," in *2014 IEEE Aerospace Conference*, pp. 1–13, Mar. 2014. ISSN: 1095-323X.

[25] Y. Zaidi, N. G. Fitz-Coy, and R. V. Zyl, "Rapid, automated, test, verification and validation for the cubesats," *International Journal of Space Science and Engineering*, vol. 5, no. 3, pp. 242–268, 2019.

[26] M. D. Grubb, "Increasing the reliability of software systems on small satellites using software-based simulation of the embedded system," Master's thesis, Graduate Theses, Dissertations, and Problem Reports, 2021.

[27] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "Fudge: fuzz driver generation at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 975–985, 2019.

[28] M. Diaz, J. Zagal, C. Falcon, M. Stepanova, J. Valdivia, M. Martinez-Ledesma, J. Diaz-Pena, F. Jaramillo, N. Romanova, E. Pacheco, *et al.*, “New opportunities offered by cubesats for space research in latin america: The suchai project case,” *Advances in Space Research*, vol. 58, no. 10, pp. 2134–2147, 2016.

[29] C. Gonzalez, C. Rojas, A. Becerra, J. Rojas, T. Opazo, and M. Diaz, “Lessons learned from building the first chilean nano-satellite : the suchai project,” in *AIAA/USU Conference on Small Satellites*, aug 2018.

[30] T. D. Cook, D. T. Campbell, and A. Day, *Quasi-experimentation: Design & analysis issues for field settings*, vol. 351. Houghton Mifflin Boston, 1979.

[31] S. T. Gutiérrez, C. I. Fuentes, and M. A. Díaz, “Introducing sost: An ultra-low-cost star tracker concept based on a raspberry pi and open-source astronomy software,” *IEEE Access*, vol. 8, pp. 166320–166334, 2020.

[32] R. Bocchino, T. Canham, G. Watney, L. Reder, and J. Levison, “F prime: an open-source framework for small-scale flight software systems,” 2018.

Appendix A

Fuzzing the SUCHAI Flight Software

Code A.1: run_experiment.py

```python
from randomsequencefuzzerwithfixedparams import RandomSequenceFuzzerWithFixedParams
from randomsequencefuzzerwithfixedparamsandexacttypes import RandomSequenceFuzzerWithFixedParamsAndExactTypes
from randomcommandsequencefuzzer import RandomCommandsSequenceFuzzer
from randomsequencefuzzer import RandomSequenceFuzzer
from flightsoftwarerunner import FlightSoftwareRunner
from subprocess import PIPE, Popen
import os
import pandas as pd
import json
import time
import argparse


def to_json(information, iterations, t, json_path):
    """
    Write information to JSON file.
    :param information: Tuple with: list of commands, list of parameters, executed commands, command results,
            commands execution time, exit code of the process, total execution time, real memory used and
            virtual memory used of each iteration.
    :param iterations: Int.
    :param t: String. Start date and time of the execution.
    :param json_path: String. Directory where to write the JSON file.
    :return:
    """
    json_lst = []
    for iteration in range(iterations):
        iter_dic = dict()

        # Add information of each command sent
        cmds_lst = []
        results = information[iteration]
        cmds_sent = results[0]
        params_sent = results[1]
        for cmd_idx in range(len(cmds_sent)):
            cmds_lst.append({"cmd_name": cmds_sent[cmd_idx], "params": params_sent[cmd_idx]})
        iter_dic['cmds'] = cmds_lst

        # Add general information of the sequence
        iter_dic['exit code'] = results[5]
        iter_dic['total time (s)'] = results[6]  # Select a sequence, then the results of it, then the time
        iter_dic['virtual memory (kb)'] = results[8]
        iter_dic['real memory (kb)'] = results[7]
        json_lst.append(iter_dic)

    # Write to json
    filename = 'data-' + t + '.txt'
    if not os.path.exists(json_path):
        os.mkdir(json_path)
    with open(json_path + filename, 'w') as outfile:
        json.dump(json_lst, outfile, indent=2, separators=(',', ': '))


def to_csv_file(information, iterations, t, csv_path):
    """
    Write information to CSV file.
    :param information: Tuple with: list of commands, list of parameters, executed commands, command results,
            commands execution time, exit code of the process, total execution time, real memory used and
            virtual memory used of each iteration.
    :param iterations: Int.
    :param t: String. Start date and time of the execution.
    :param csv_path: String. Directory where to write the CSV file.
    :return:
    """
    csv_lst = []
    for iteration in range(iterations):
        # Add information of each command sent
        csv_lst.append([])
        results = information[iteration]
        cmds_sent = results[0]
        params_sent = results[1]
        for j in range(len(cmds_sent)):
            csv_lst[iteration].append(cmds_sent[j])
            csv_lst[iteration].append("'" + params_sent[j] + "'")

        # Add general information of the sequence
        csv_lst[iteration].append(results[5])
        csv_lst[iteration].append(results[6])
        csv_lst[iteration].append(results[8])
        csv_lst[iteration].append(results[7])

    cols = []
    number_of_commands = len(information[0][0])

    # Add columns
    for cmd_idx in range(number_of_commands):
        cols.append("Command")
        cols.append("Parameters")
    cols.append("Exit Code")
    cols.append("Total Time")
    cols.append("Virtual Memory (kB)")
    cols.append("Real Memory (kB)")

    # Create dataframe and write to CSV file
    information_df = pd.DataFrame(csv_lst, columns=cols)
    filename = 'data-' + t + '.csv'
    if not os.path.exists(csv_path):
        os.mkdir(csv_path)
    information_df.to_csv(csv_path + filename, index=False)


def run_experiment(random_fuzzer, iterations=10, cmds_number=10, csv_path='', json_path=''):
    """
    Create a random fuzzer instance to execute flight software with random input.
    :param random_fuzzer: Class. Fuzzer.
    :param iterations: Int.
    :param cmds_number: Int. Number of commands to execute each iteration.
    :param csv_path: String. Directory for CSV reports. The directory must exist. Must end with a "/" character.
    :param json_path: String. Directory for JSON reports. The directory must exist. Must end with a "/" character.
    :return:
    """
    print("Commands number: " + str(cmds_number) + ", iteration: " + str(iterations))

    # print(params_type)
    # Run zmqhub.py (ipc)
    # ex_zmqhub = Popen(["python3", "zmqhub.py", "--ip", "/tmp/suchaifs", "--proto", "ipc"], stdin=PIPE)
    # Run zmqhub.py (tcp)
    ex_zmqhub = Popen(["python3", "zmqhub.py", "--mon"], stdin=PIPE)

    # Set variables
    exec_dir = "../../Git/suchai-flight-software4/build_groundstation/"
    exec_cmd = "./SUCHAI_Flight_Software"

    # Run flight software sending n_cmds random commands with 1 random parameter
    prev_dir = os.getcwd()
    os.chdir(exec_dir)
    start_time = time.strftime("%Y%m%d-%H%M%S")  # Measure start time to include it in the report name
    outcomes = random_fuzzer.runs(FlightSoftwareRunner(exec_cmd=exec_cmd), iterations)
    os.chdir(prev_dir)

    # Kill zmqhub.py
    ex_zmqhub.kill()

    # Write outcome information report
    to_json(outcomes, iterations, start_time, json_path)

    # Write report to csv file
    to_csv_file(outcomes, iterations, start_time, csv_path)


def get_parameters():
    """
    Parse script arguments.
    Every path argument must end with a "/".
    """
    parser = argparse.ArgumentParser(prog='run_experiment.py')

    parser.add_argument('--csv_path', type=str, default='Dummy-Folder/CSV/', help="Save CSV reports in this directory")
    parser.add_argument('--json_path', type=str, default='Dummy-Folder/JSON/', help="Save JSON reports in this directory")
    parser.add_argument('--time_path', type=str, default='Dummy-Folder/Time/', help="Save time reports in this directory")
    parser.add_argument('--iterations', nargs='+', type=int, default="10 100 500 1000", help="Number of sequences")
    parser.add_argument('--commands_number', nargs='+', type=int, default="5 10 50 100", help="Number of commands in a "
                                                                                                  "sequence")
    parser.add_argument('--min_length', type=int, default=0, help="Minimum length of the random command names.")
    parser.add_argument('--max_length', type=int, default=10, help="Maximum length of the random command names.")
    parser.add_argument('--char_start', type=int, default=33, help="Index of the range that indicates where to start "
                                                                    "producing random command names in ASCII code.")
    parser.add_argument('--char_range', type=int, default=93, help="Length of the characters range in ASCII code.")
    parser.add_argument('--strategy', type=int, default=0, help="Number of the strategy to be run")
    parser.add_argument('--commands_file', type=str, default='suchai_cmd_list_all.csv', help="Filename with the SUCHAI "
                                                                                "Flight Software commands "
                                                                                "and parameters type.")

    return parser.parse_args()


def main(time_path, csv_path, json_path, iterations, commands_number, min_length, max_length, char_start, char_range, fuzz_class,
        ↪ commands_file):
    """
    :param time_path: Directory for time reports. The directory must exist. Must end with a "/" character.
    :param csv_path: Directory for CSV reports. The directory must exist. Must end with a "/" character.
    :param json_path: Directory for JSON reports. The directory must exist. Must end with a "/" character.
    :param iterations: List. Each element represents a sequences' number.
    :param commands_number: List. Each element represents a commands' number in a sequence.
    :param min_length: Int. Minimum length of the random command names.
    :param max_length: Int. Maximum length of the random command names.
    :param char_start: Int. Index of the range that indicates where to start producing random command names in ASCII code.
    :param char_range: Int. Length of the characters range in ASCII code.
    :param fuzz_class: Class. Fuzzer class to be used.
    :param commands_file: String. Filename with the SUCHAI Flight Software commands and parameters type.
    :return:
    """
    # Create file to write execution time for each iteration
    curr_time = time.strftime("%Y%m%d-%H%M%S")

    if not os.path.exists(time_path):
        os.mkdir(time_path)

    f = open(time_path + 'exec_time-' + curr_time + '.txt', '+w')
    f.close()

    # Run experiment and add execution time of each iteration to time reports
    for num_cmds in commands_number:

        # Create fuzzer instance
        fuzzer = fuzz_class(commands_file, min_length=min_length, max_length=max_length, char_start=char_start,
                            char_range=char_range, n_cmds=num_cmds)

        for iter in iterations:
            exec_start_time = time.time()
            run_experiment(fuzzer, int(iter), int(num_cmds), csv_path, json_path)
            with open(time_path + 'exec_time-' + curr_time + '.txt', 'a') as f:
                f.write("%s\n" % (time.time() - exec_start_time))


if __name__ == "__main__":
    args = get_parameters()
    strategies_fuzz_classes = {0: RandomCommandsSequenceFuzzer,
                               1: RandomSequenceFuzzer,
                               2: RandomSequenceFuzzerWithFixedParams,
                               3: RandomSequenceFuzzerWithFixedParamsAndExactTypes}
    main(args.time_path, args.csv_path, args.json_path, args.iterations, args.commands_number, args.min_length, args.max_length, args.
        ↪ char_start, args.char_range, strategies_fuzz_classes[args.strategy], args.commands_file)
```

Code A.2: randomcommandsequencefuzzer.py

```
from fuzzingbook.Fuzzer import RandomFuzzer
from flightsoftwarerunner import *


class RandomCommandsSequenceFuzzer(RandomFuzzer):
    def __init__(self, commands_filename, min_length=10, max_length=100, char_start=0, char_range=127, n_cmds=1):
        RandomFuzzer.__init__(self, min_length, max_length, char_start, char_range)
        self.n_cmds = n_cmds
        self.commands_file = commands_filename

    def run(self, runner=FlightSoftwareRunner()):
        """
        Run 'runner' with fuzzed parameters and random commands chosen from a list
        :param runner:
        :return: Results obtained from running the program with fuzzed input
        """
        cmds_to_send = []
        params_to_send = []
        for i in range(0, self.n_cmds):
            cmds_to_send.append(self.fuzz())
            params = ""
            params_to_send.append(params)
        return runner.run_process(cmds_to_send, params_to_send)
```

Code A.3: randomsequencefuzzer.py

```
from fuzzingbook.Fuzzer import RandomFuzzer
from flightsoftwarerunner import *
import random

MIN_INT = -2147483648
MAX_INT = 2147483647
MIN_LONG = -9223372036854775808
```

```python
MAX_LONG = 9223372036854775807
MAX_U_INT = 18446744073709551615
MIN_FLOAT = −3.402823e+38
MAX_FLOAT = 3.402823e+38


class RandomSequenceFuzzer(RandomFuzzer):
    def __init__(self, commands_filename, min_length=10, max_length=100,
                 char_start=33, char_range=93, n_cmds=1):
        RandomFuzzer.__init__(self, min_length, max_length, char_start, char_range)
        self.n_cmds = n_cmds
        self.commands_file = commands_filename
        self.fs_cmds = []
        self.get_commands_names(self.commands_file)
        self.fuzz_funcs = {}

    def get_commands_names(self, commands_list):
        """
        Get command names list and set into the fs_cmds variable.
        :param commands_list: String. File name of the SUCHAI flight software commands.
        :return:
        """
        commands_names = []
        with open(commands_list) as file_list:
            for row in file_list:
                commands_names.append(row.split(', ')[0])
        self.fs_cmds = commands_names

    def fuzz_int(self):
        """
        Produce random integer.
        :return: String. Random integer converted to string.
        """
        # Min. length and max. length are not considered
        return str(random.randint(MIN_INT, MAX_INT))

    def fuzz_long(self):
        """
        Produce random long.
        :return: String. Random long converted to string.
        """
        # Min. length and max. length are not considered
        return str(random.randint(MIN_LONG, MAX_LONG))

    def fuzz_unsigned_int(self):
        """
        Produce random unsigned int.
        :return: String. Random unsigned int converted to string.
        """
        # Min. length and max. length are not considered
        return str(random.randint(0, MAX_U_INT))

    def fuzz_float(self):
        """
        Produce random float.
        :return: String. Random float converted to string.
        """
        # Min. length and max. length are not considered
        return str(random.uniform(MIN_FLOAT, MAX_FLOAT))

    def fuzz_string(self):
        """
        Produce random string between self.min_length and self.max_length size.
        :return: String.
        """
        string_length = random.randrange(self.min_length, self.max_length + 1)
        out = ""
        for i in range(0, string_length):
            out += chr(random.randrange(self.char_start, self.char_start + self.char_range))
        return out

    def generate_seqs(self, iterations):
        """
        Generate random sequences.
        :param iterations:
        :return: List. List of sequences created (commands and parameters).
        """
        self.fuzz_funcs = [self.fuzz_int, self.fuzz_float, self.fuzz_long, self.fuzz_unsigned_int, self.fuzz_string]
        sequences = []
        for iter in range(iterations):
            seq = []
            for i in range(0, self.n_cmds):
                cmd = random.choice(self.fs_cmds)
                n_params = random.randint(0, 11) # 11 is the max number of params that a cmd has
                fuzz_to_apply = [random.choice(self.fuzz_funcs) for i in range(n_params)]
                params = [fuzz_to_apply[i]() for i in range(n_params)]
                params = " ".join(params)
                seq.append(cmd + " " + params)
            sequences.append(seq)
        return sequences

    def run(self, runner=FlightSoftwareRunner()):
        """
        Run 'runner' with fuzzed parameters and random commands chosen from a list.
        :param runner: Class. Runner.
        :return: Results obtained from running the program with fuzzed input.
        """
        self.fuzz_funcs = [self.fuzz_int, self.fuzz_float, self.fuzz_long, self.fuzz_unsigned_int, self.fuzz_string]
        cmds_to_send = []
        params_to_send = []
        print(self.fs_cmds)
        for i in range(0, self.n_cmds):
            cmds_to_send.append(random.choice(self.fs_cmds))
            n_params = random.randint(0, 11)  # 11 is the max number of params that a cmd has
            fuzz_to_apply = [random.choice(self.fuzz_funcs) for i in range(n_params)]
            params = [fuzz_to_apply[i]() for i in range(n_params)]
            params = " ".join(params)
            params_to_send.append(params)
        return runner.run_process(cmds_to_send, params_to_send)
```

Code A.4: randomsequencefuzzerwithfixedparams.py

```python
from randomsequencefuzzer import *


class RandomSequenceFuzzerWithFixedParams(RandomSequenceFuzzer):
    def __init__(self, commands_filename, min_length=10, max_length=100,
                 char_start=32, char_range=32, n_cmds=1):
        RandomSequenceFuzzer.__init__(self, commands_filename, min_length, max_length, char_start, char_range, n_cmds)
        self.number_of_params = []
        self.get_parameters_numbers(self.commands_file)

    def get_parameters_numbers(self, commands_list):
        """
        Get list of the number of parameters each command of the SUCHAI flight software receives and set into the
        variable "number_of_params"
        :param commands_list: String. File name of the SUCHAI flight software commands.
        :return:
        """
        parameters_numbers = []
        with open(commands_list) as file_list:
            for row in file_list:
                parameters_numbers.append(int(row.split(', ')[1]))
        self.number_of_params = parameters_numbers

    def run(self, runner=FlightSoftwareRunner()):
        """
        Run 'runner' with fuzzed parameters and random commands chosen from a list
        :param runner: Class. Runner.
        :return: Results obtained from running the program with fuzzed input
        """
        self.fuzz_funcs = [self.fuzz_int, self.fuzz_float, self.fuzz_long, self.fuzz_unsigned_int, self.fuzz_string]
        cmds_to_send = []
        params_to_send = []
        print(self.fs_cmds)
        for i in range(0, self.n_cmds):
            ind_chosen = random.randint(0, len(self.fs_cmds) - 1)
            cmds_to_send.append(self.fs_cmds[ind_chosen])
            n_params = self.number_of_params[ind_chosen]
            fuzz_to_apply = [random.choice(self.fuzz_funcs) for i in range(n_params)]
            params = [fuzz_to_apply[i]() for i in range(n_params)]
            params = " ".join(params)
            params_to_send.append(params)
        return runner.run_process(cmds_to_send, params_to_send)
```

Code A.5: randomsequencefuzzerwithfixedparamsandexacttypes.py

```python
from randomsequencefuzzer import *


class RandomSequenceFuzzerWithFixedParamsAndExactTypes(RandomSequenceFuzzer):
    def __init__(self, commands_filename, min_length=10, max_length=100,
                 char_start=32, char_range=32, n_cmds=1):
        RandomSequenceFuzzer.__init__(self, commands_filename, min_length, max_length, char_start, char_range, n_cmds)
        self.params_types = []
        self.get_parameters_types(self.commands_file)

    def get_parameters_types(self, commands_list):
        """
        Get list of the types of parameters each command of the SUCHAI flight software receives and set into the
        variable "params_types"
        :param commands_list: String. File name of the SUCHAI flight software commands.
        :return:
        """
        parameters_types = []
        with open(commands_list) as file_list:
            for row in file_list:
                params_str_without_newline = row.rstrip('\n')
                parameters_types.append(params_str_without_newline.split(', ')[2:])
        self.params_types = parameters_types

    def run(self, runner=FlightSoftwareRunner()):
        """
        Run 'runner' with fuzzed parameters and random commands chosen from a list
        :param runner: Class. Runner.
        :return: Results obtained from running the program with fuzzed input
        """
        self.types_dic = {"%d": self.fuzz_int, "%i": self.fuzz_int, "%f": self.fuzz_float, "%ld": self.fuzz_long,
                          "%u": self.fuzz_unsigned_int, "%s": self.fuzz_string, "%n": self.fuzz_string,
                          "%p": self.fuzz_string, }
        cmds_to_send = []
        params_to_send = []
        print(self.fs_cmds)
        for i in range(0, self.n_cmds):
            ind_chosen = random.randint(0, len(self.fs_cmds) - 1)
            cmds_to_send.append(self.fs_cmds[ind_chosen])  # Append the command
            n_params = len(self.params_types[ind_chosen])  # Determine parameters number of the command
            fuzz_to_apply = [self.types_dic[self.params_types[ind_chosen][j]] for j in range(n_params)]  # Choose fuzz function for each
        ↪ parameter type of the command
            params = [fuzz_to_apply[i]() for i in range(n_params)]  # Run each fuzz function
            params = " ".join(params)
            params_to_send.append(params)
        return runner.run_process(cmds_to_send, params_to_send)
```

Code A.6: flightsoftwarerunner.py

```
from fuzzingbook.Fuzzer import Runner
from subprocess import Popen, PIPE
from fuzzcspzmqnode import *
from proc_info import *
import time


class FlightSoftwareRunner(Runner):
    def __init__(self, exec_cmd="./SUCHAI_Flight_Software"):
        self.exec_cmd = exec_cmd

    def run_process(self, cmds_list=[], params_list=[]):
        """
        Runs SUCHAI flight software and send commands to it until the process is done.
        :param cmds_list:
        :param params_list:
        :return: Tuple. List of commands executed, list of results, execution time and memory usage.
        """
        # Each element of params_list matches with a command from the commands list
        assert len(cmds_list) == len(params_list), "Each sequence of parameters must match with a command"

        # Send commands to the flight software through zmq
        dest = "1"
        addr = "9"
        port = "12"
        #ipc
        #node = FuzzCspZmqNode(addr, hub_ip="/tmp/suchaifs", proto="ipc")
        #tcp
        node = FuzzCspZmqNode(addr)
        node.start()

        # Execute flight software
        time.sleep(1)
        init_time = time.time()  # Start measuring execution time of the sequence
        suchai_process = Popen([self.exec_cmd], stdin=PIPE)
        time.sleep(4)

        # Clean database
        print("node send: drp_ebf 1010")  # For debugging purposes
        header = CspHeader(src_node=int(addr), dst_node=int(dest), dst_port=int(port), src_port=55)
        node.send_message("drp_ebf 1010", header)

        # Start sending random commands
        for i in range(0, len(cmds_list)):
            # time.sleep(0.5)  # Give some time to zmqnode threads (writer and reader)
            cmd = cmds_list[i]
            params = params_list[i]
            print("node send:", cmd + " " + params)  # For debugging purposes
            header = CspHeader(src_node=int(addr), dst_node=int(dest), dst_port=int(port), src_port=55)
            node.send_message(cmd + " " + params, header)

        # Get memory usage of the SUCHAI process
        proc_pid = suchai_process.pid
        vm, rm = get_mem_info(proc_pid)

        # Exit SUCHAI process
        hdr = CspHeader(src_node=int(addr), dst_node=int(dest), dst_port=int(port), src_port=56)
        node.send_message("obc_reset", hdr)

        # Get SUCHAI process return code
        return_code = suchai_process.wait()
        end_time = time.time()  # End measuring execution time of the sequence
        print("Return code: ", return_code)  # For debugging purposes

        # Get commands, results, execution time and memory usage
        executed_cmds = node.filter_cmds_names()
        results = node.filter_results()
        cmds_time = node.filter_cmds_exec_time()
        total_exec_time = end_time - init_time

        node.stop()
        return cmds_list, params_list, executed_cmds, results, cmds_time, return_code, total_exec_time, rm, vm
```

Code A.7: fuzzcspzmqnode.py

```
from zmqnode import *


class FuzzCspZmqNode(CspZmqNode):
    def __init__(self, node, hub_ip='localhost', in_port="8001", out_port="8002", reader=True, writer=True, proto="tcp"):
        """
        :param node:
        :param hub_ip:
        :param in_port:
        :param out_port:
        :param monitor:
        :param console:
        """
        CspZmqNode.__init__(self, node, hub_ip, in_port, out_port, reader, writer, proto)
        self.all_messages_queue = Queue()
        self.init_ready_queue = Queue()
        self.messages_list = []

    def read_message(self, message, header=None):
        """
        Put all received messages on a queue. This method is called from the _reader thread.
        :param message: Str. Message received.
        :param header: CspHeader. CSP header.
        :return:
        """
        self.all_messages_queue.put([message.decode('ASCII', 'ignore')])

    def messages_queue_to_list(self):
        """
        Save in list all the messages from the messages queue.
        :return:
        """
        while not self.all_messages_queue.empty():
            self.messages_list.extend(self.all_messages_queue.get())

    def print_messages(self):
        """
        Print all the received messages as list.
        :return:
        """
        print("Full messages list:")
        print(self.messages_list)

    def filter_cmds_names(self):
        """
        Filter messages indicating that a command is being run and save its name.
        :return: Names of the command list.
        """
        if not self.messages_list:
            self.messages_queue_to_list()
        self.print_messages()
        cmds_names = [cmd.split()[3] for cmd in self.messages_list if 'Running the command' in cmd]
        print("Names of commands:")  # For debugging purposes
        print(cmds_names)  # For debugging purposes

        return cmds_names

    def filter_results(self):
        """
        Filter messages indicating a command result and save it.
        :return: List. Results of the commands sent.
        """
        if not self.messages_list:
            self.messages_queue_to_list()

        results = [result.split()[2] for result in self.messages_list if 'Command result' in result]
        print("Results")  # For debugging purposes
        print(results)  # For debugging purposes

        return results

    def filter_cmds_exec_time(self):
        """
        Filter messages indicating execution time of the commands and save it.
        :return: List. Commands time execution.
        """
        if not self.messages_list:
            self.messages_queue_to_list()

        cmds_time = [time.split()[-1] for time in self.messages_list if 'Command result' in time]
        print("Time")  # For debugging purposes
        print(cmds_time)  # For debugging purposes

        return cmds_time
```

Appendix B

Fuzzing the FPrime Software

Code B.1: main.py

```python
from seq_fuzzer import RandomSequenceFuzzer
from fprimegdsrunner import FprimeGDSRunner
import time
from selenium.webdriver import Firefox
import matplotlib.pyplot as plt
import argparse
import json
import os


def to_json(outcomes, total_time, path):
    """
    Write execution information to JSON file.
    :param outcomes: List. Contains (seq, total_time, relevant_acts) tuple. This is the execution info.
    :param total_time: Int. Total execution time of the entire process.
    :param path: Str. Main directory that contains JSON files.
    :return:
    """
    json_dict = {}

    # Sequence information
    json_list = []
    number_of_seqs = len(outcomes)
    for sequence in outcomes:
        sequence_dict = dict()

        # Add information of each command sent
        commands_list = []
        commands_info = sequence[0]
        number_of_cmds = len(commands_info)
        user_time = sequence[1]
        relevant_events = sequence[2]
        commands_names = [command_params[0] for command_params in commands_info]
        parameters = [command_params[1:] for command_params in commands_info]

        for i in range(len(commands_names)):
            cmd_dict = dict()
            params_dict = {}
            cmd_dict["cmd_name"] = commands_names[i]
            for param in parameters[i]:
                param_name = param[0]
                param_val = param[1]
                params_dict[param_name] = param_val
            cmd_dict["params"] = params_dict
            commands_list.append(cmd_dict)
        sequence_dict["cmds"] = commands_list
        sequence_dict["user time"] = user_time
        sequence_dict["relevant events"] = relevant_events
        json_list.append(sequence_dict)
    print(json_list)
    json_dict["sequence information"] = json_list

    # Add information of the entire process
    exec_dict = {}
    exec_dict["total time"] = total_time
    json_dict["total execution information"] = exec_dict

    # Write to json
    number_of_seqs_dir = path + str(number_of_seqs)
    number_of_commands_str = str(number_of_cmds) + "_cmds"
    filename = "/" + number_of_commands_str + "-" + time.strftime("%Y%m%d_%H%M%S") + '.txt'
    json_path = number_of_seqs_dir

    if not os.path.exists(json_path):
        os.makedirs(json_path)
    with open(json_path + filename, 'w') as outfile:
        json.dump(json_dict, outfile, indent=2, separators=(',', ': '))


def run_experiment(random_fuzzer, n_seqs):
    """
    Create a random fuzzer instance to execute flight software with random input.
    :param random_fuzzer: Class. Fuzzer.
    :param n_seqs: Int. Number of sequences.
    :return: process_time_list: List. Contains (seq, total_time, relevant_acts) tuple. This is the execution info.
    """
    driver = Firefox()
    driver.get("http://127.0.0.1:5000/")
    process_time_list = random_fuzzer.runs(FprimeGDSRunner(driver), n_seqs)
    driver.quit()
    return process_time_list


def get_parameters():
    """
    Parse script arguments.
    Every path argument must end with a "/".
    """
    parser = argparse.ArgumentParser(prog='run_experiment.py')

    parser.add_argument('--json_path', type=str, default='results/', help="Save JSON reports in this directory")
    parser.add_argument('--n_seqs', type=int, default=250, help="Number of sequences")
    parser.add_argument('--n_cmds', type=int, default=5, help="Number of commands in a sequence")
    parser.add_argument('--min_length', type=int, default=1, help="Minimum length of the random command names.")
    parser.add_argument('--max_length', type=int, default=21, help="Maximum length of the random command names.")
    parser.add_argument('--char_start', type=int, default=33, help="Index of the range that indicates where to start "
                                                                "producing random command names in ASCII code.")
    parser.add_argument('--char_range', type=int, default=93, help="Length of the characters range in ASCII code.")

    return parser.parse_args()


def main(json_path, n_seqs, n_cmds, min_length, max_length, char_start, char_range):
    """
    :param json_path: Str. Directory for JSON reports. The directory must exist. Must end with a "/" character.
    :param n_seqs: Int. Number of sequences.
    :param n_cmds: Int. Number of commands per sequence.
    :param min_length: Int. Minimum length of the random command names.
    :param max_length: Int. Maximum length of the random command names.
    :param char_start: Int. Index of the range that indicates where to start producing random command names in ASCII code.
    :param char_range: Int. Length of the characters range in ASCII code.
    :return:
    """
    random_fuzzer = RandomSequenceFuzzer(min_length=min_length, max_length=max_length, char_start=char_start,
                                         char_range=char_range, seq_size=n_cmds)

    time_exec_init = time.time()
    outcome_list = run_experiment(random_fuzzer, n_seqs)
    total_exec_time = time.time() - time_exec_init

    # Write outcome information report
    to_json(outcome_list, total_exec_time, json_path)


if __name__ == "__main__":
    if __name__ == "__main__":
        args = get_parameters()
        main(args.json_path, args.n_seqs, args.n_cmds, args.min_length, args.max_length, args.char_start, args.char_range)
```

Code B.2: seq_fuzzer.py

```python
from fuzzingbook.Fuzzer import RandomFuzzer
from fprimegdsrunner import FprimeGDSRunner
import random
import requests
import json


class RandomSequenceFuzzer(RandomFuzzer):
    def __init__(self, min_length=10, max_length=100,
                 char_start=32, char_range=32, seq_size=1):
        RandomFuzzer.__init__(self, min_length, max_length, char_start, char_range)
        self.n_cmds = seq_size

        # Get list of available commands
        cmds_json_url = "http://127.0.0.1:5000/dictionary/commands"
        headers = {
            'User-Agent': 'Mozilla/5.0 (X11; Linux x86_64; rv:89.0) Gecko/20100101 Firefox/89.0'
        }
        html_response = requests.get(cmds_json_url, headers)
        self.cmds_json = json.loads(html_response.text)
        self.avail_cmds = list(self.cmds_json.keys())
        self.avail_cmds.remove('pingRcvr.PR_StopPings')

    def generate_seq(self):
        """
        Generates a random commands' sequence. It requires interaction with the web browser UI elements.
        :return: seq: List. Contains the random commands and parameters generated.
        """
        random.seed()
        seq = []

        # Iterate through elements in a sequence. An element is a particular command and its parameters.
        for i in range(self.n_cmds):
            cmd_params = []
            # Get number of commands available
            n_cmds = len(self.avail_cmds)

            # Choose command index
            random_ind = random.randint(0, n_cmds-1)

            # Get command name by index
            cmd_name = self.avail_cmds[random_ind]

            # Add to list
            cmd_params.append(cmd_name)

            # Get number of params
            # NOTE: ALL PARAMETERS WITH TYPE != ENUM MUST BE ADDED AS A STRING
            for arg in self.cmds_json[cmd_name]["args"]:
                # If param is Enum type, choose random index
                if arg["type"] == 'Enum':
                    # Choose random index
                    possible_options = len(arg["possible"])
                    random_opt = random.randint(0, possible_options-1)
                    # Create tuple (<parameter name>, <random index value>)
                    param_opt = (arg["name"], random_opt)
                # Else, create tuple (<parameter name>, <random string value>)
                else:
                    param_opt = (arg["name"], self.fuzz())
                cmd_params.append(param_opt)
            seq.append(cmd_params)
        return seq

    def run(self, runner=FprimeGDSRunner):
        """
        Run 'runner' with fuzzed sequence of commands and parameters.
        :param runner: Class.
        :return: seq_info: Tuple. Contains information of each sequence and the entire process execution.
        """
        seq = self.generate_seq()
        total_time, relevant_acts = runner.run_process(seq)
        seq_info = (seq, total_time, relevant_acts)
        print("seq_info: ", seq_info)
        return seq_info
```

Code B.3: fprimegdsrunner.py

```
from fuzzingbook.Fuzzer import Runner
import time


class FprimeGDSRunner(Runner):
    def __init__(self, driver):
        """Initialize"""
        Runner.__init__(self)
        self.driver = driver

    def run_process(self, seq):
        """
        Sends commands from F Prime GDS UI to the Ref application of FPrime software. It also requires interaction with
        the web browser UI elements.
        :param seq: List. Contains the random commands and parameters generated.
        :return: (total_time, relevant_acts): Tuple. Information of the sequence execution: time and found high severity
        events with their corresponding description.
        """
        init_time = time.time()
        print("Sending sequence: ")
        print(seq)
        for elem in seq:
            time.sleep(1)
            # Click dropdown list
            dropdown = self.driver.find_elements_by_id("mnemonic")[0]
            dropdown.click()
            # Get elements list
            time.sleep(1)
            li_elems = self.driver.find_elements_by_tag_name("li")
            li_names = [element.text for element in li_elems]

            #Debug ValueError: name is not in list
            print("li_elems: ")
            print(li_elems)
            print("li_names: ")
            print(li_names)
            # Select command
            cmd_index = li_names.index(elem[0])
            cmd = li_elems[cmd_index]
            cmd.click()

            # Iterate through parameters. NOTE: the first element is the command
            if len(elem) > 1:
                for i in range(1, len(elem)):
                    arg_name = elem[i][0]
                    arg_val = elem[i][1]
                    selection_element = self.driver.find_element_by_id(arg_name)
                    # The argument value must be chosen through a dropdown list
                    if type(arg_val) == int:
                        # Display argument dropdown list
                        selection_element.click()
                        # Click chosen option
                        all_options = self.driver.find_elements_by_class_name("vs__dropdown-option")
                        option = all_options[arg_val]
                        option.click()
                    # The argument value must be fed as input
                    else:
                        # Write input in the parameter field
                        selection_element.send_keys(arg_val)
            # Send command and parameters
            send_button = self.driver.find_elements_by_class_name("col-2.btn.btn-primary")[0]
            send_button.click()
        time.sleep(2)
        # Click events tab to get status
        events_tab = li_elems[li_names.index("Events")]
        events_tab.click()

        # Check if there is any fatal status
        table = self.driver.find_elements_by_class_name("sortable.table.table-bordered.table-hover")
        table_body = table[2].find_element_by_xpath(".//tbody")

        relevant_acts = []
        for row in table_body.find_elements_by_xpath(".//tr"):
            severity = row.find_elements_by_xpath(".//td")[3].text
            if "FATAL" in severity or "WARNING_HI" in severity:
                relevant_acts.append((severity, row.find_elements_by_xpath(".//td")[4].text))

        # Clear table
        clear_button = self.driver.find_elements_by_class_name("col-3.btn.btn-secondary")[0]
        clear_button.click()

        # Return to commanding tab
        events_tab = li_elems[li_names.index("Commanding")]
        events_tab.click()

        total_time = time.time() - init_time

        return total_time, relevant_acts
```