IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received July 8, 2021, accepted July 30, 2021, date of publication August 12, 2021, date of current version August 23, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3104283

Systematic Fuzz Testing Techniques on a Nanosatellite Flight Software for Agile Mission Development

TAMARA GUTIERREZ[1], ALEXANDRE BERGEL[1], CARLOS E. GONZALEZ[2], CAMILO J. ROJAS[2], AND MARCOS A. DIAZ[2], (Member, IEEE)

[1]Intelligent Software Construction Laboratory (ISCLab), Department of Computer Science (DCC), Faculty of Physical and Mathematical Sciences, University of Chile, Santiago 8370448, Chile

[2]Space and Planetary Exploration Laboratory (SPEL), Electrical Engineering Department, Faculty of Physical and Mathematical Sciences, University of Chile, Santiago 8370448, Chile

Corresponding author: Tamara Gutierrez (tamara.gutierrez@ug.uchile.cl)

This work was supported in part by Lam Research, in part by the ANID Fondecyt Regular 1221907 and 1200067, in part by Fondecyt 1151476, in part by Anillo ACT1405, in part by CONICYT-PCHA/Doctorado Nacional/2016-21161016, in part by the Force Office of Scientific Research (AFOSR) under Award FA9550-18-1-0249 and Award FA9550-20-1-0303, and in part by the CONICYT QUIMAL 190004.

ABSTRACT The success of CubeSat space missions depends on the ability to perform properly in a harsh environment. A key component in space missions is the flight software, which manages all of the processes executed by the satellite on its onboard computer. 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. Extensive operation testing is the primary technique used by CubeSats developers to ensure flight software quality and avoid such failures. The ‘‘New Space’’ requirements pressure to add ‘‘agility’’ to the software development, which could limit the capacity to test. While advanced and beneficial software testing techniques are found in the software engineering field, CubeSat software solutions mostly rely on unit testing, software in the loop simulation, and hardware in the loop simulation. In this work, fuzz testing techniques were developed, implemented, and evaluated as a manner to expedite operational testing of CubeSats while maintaining their completeness. The impact of the tools was evaluated by using the three new 3U CubeSats under development at the University of Chile. We identified twelve bugs not covered by classic testing strategies in less than three days. These failures were reported, fixed, and characterized by the developers in eight sprint sessions. Our results indicate that fuzz testing improved the completeness of flight software testing through automation and with almost no development interruption. Although our approach has been tested on the SUCHAI flight software, it applies to systems that follow a similar architecture.

INDEX TERMS CubeSat, embedded software, flight software, nanosatellites, testing, fuzz testing, software quality, open source.

I. INTRODUCTION

The first conception of a CubeSat nanosatellite prototype came up only 20 years ago approximately. Initially, nanosatellites were conceived with a mainly educational purpose in which students are able to experience the development and operation of a satellite in the time frame of a college degree [1]. Nowadays, nanosatellites have opened several opportunities but still need to overcome multiple challenges to reach their full potential [2]. Nanosatellites increasingly require more attention to their quality attributes to be successful in more complex missions. Specifically, the flight software of nanosatellites is a critical factor in determining a satellite's quality because it controls most of the tasks that must be executed once in space. The success rate of a space mission is highly dependent on the quality of its flight software [3].

In the space field, several testing techniques are used to assess flight software quality. 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 state of the art, the most

The associate editor coordinating the review of this manuscript and approving it for publication was Michele Magno.

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, the test cases must be predefined because these techniques are difficult 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]. The ability to implement 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.

Currently, the high expectations of CubeSats are based on the possibility of developing a large number of satellites (mega-constellations) in a cost-effective manner [12], [13]. The cost-effectiveness requires that these constellations can be developed by small inexperienced groups (e.g., startups) in short development cycles. This commercial hardware usually has more computing power with less power consumption, is more miniaturized and up-to-date regarding technological needs. However, this hardware and the software that controls it has almost no flight heritage, making them risky to use in space. Testing automation arises as to the most cost-effective manner of keeping agile development while ensuring the required quality and robustness for the spacecraft.

Fuzz testing is an automated software testing technique that consists in automatically generating random input to find software vulnerabilities [14]. In need of looking for an automatable and agile software testing technique applicable to nanosatellites, we study the usage of fuzz testing in the SUCHAI nanosatellite flight software [10]. Thanks to its design, 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 composition of commands' characters, and the composition of parameters' characters give rise to four proposed strategies defined in Section IV.

A. CONTRIBUTIONS AND RESULTS

This article 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 article are:

- Presents a methodology we have developed to apply fuzz testing to nanosatellites' flight software as part of an agile CubeSat flight software methodology;
- Highlights and discusses the challenges we faced and describes the main requirements to implement this technique in similar projects;
- Presents a compelling case study of applying modern testing techniques to a critical embedded software which, we believe, opens a niche in the field of nanosatellite flight software testing.

As a result, fuzz testing has proven to be very valuable in our situation as we discovered: (i) various potential software failures, whose severity ranged from middle to severe, (ii) identified a sequence of commands to trigger and reproduce these failures, and (iii) addressed these software failures. We provided the necessary detail of our approach, hoping other researchers in the field of flight software development will benefit from our effort and results.

B. SCIENTIFIC SCOPE

This article is essentially based on the experience we have gained by developing the flight software of a series of nanosatellites (SUCHAI-I, SUCHAI-II, SUCHAI-III, and PlantSat) and its subsystems/payloads. This experience indicates to us that preparing the flight software for larger assembly lines may be challenging and requires agility and automation regarding testing to achieve the desired robustness. However, testing flight software is still an incipient field. Currently, only sporadic experiences have been reported, and no dedicated low-cost testing practices have been proposed thus far. Flight software is a highly valuable component, and techniques to improve its robustness deserve to be carefully studied and disseminated.

Whereas the area of software engineering has produced many techniques, including fuzz testing, there is no public report of its utilization on CubeSats' flight software. Our contributions improve the testing practices of flight software, which currently appear to be conducted in a non-automatic way. Our observations from different research agencies developing flight software highlight a gap between the way flight software is developed and the techniques proposed by the software engineering community. We expect that our experience and proposed methodology could contribute to reducing this gap.

C. OUTLINE

This article is organized as follows: Section II presents the work related to our effort; Section III gives the context of this article by describing the SUCHAI flight software; Section IV details the methodology we have developed to apply fuzz testing to the SUCHAI flight software; Section V presents the results of our methodology; Section VI lists the threats to the validity of our experiment and analyzes its applicability to other flight software; Section VII presents the main conclusions of this work and highlights open issues to address in future works.

II. RELATED WORK

The most common testing techniques for CubeSats found in the literature are directly attached to hardware testing. Kiesbye *et al.* [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 the simulation environment was used 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* [15] 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 that were applied to the software were strongly based on the requirements and documentation. Thanks to the application of testing methodologies to the flight software, most bugs were found in the early stages of the development process. This begins with requirement analysis. After this stage, the flight software team formulated a test strategy and began the test planning. After the test cases generation, scripting, and execution, each bug found was reported. 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 is an example of how arduous testing is for small groups developing CubeSats in an academic environment. The need for time planning and agility in the process of software development and testing is crucial to produce a reliable system, especially in groups with those attributes.

Johl *et al.* [16] present a reusable command and data handling (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 of this work affirm that flight software testing is an integral step in the development process. Therefore, to validate it, white-box and black-box testing techniques were planned and applied. The testing technique applied to the C&DH system was unit testing. Command execution testing and day-in-the-life testing were proposed to be applied as future work [16]. Day in-the-life testing refers to verifying the functionality of the fully integrated satellite while a sequence of operations is being executed. We identify this type of testing as 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.* [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 very 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.* [17] 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, which is 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 HIL and SILS.

Other concepts like software portability and rigorous software design are also present in the current related work and have been a topic of discussion because of the recent rise of CubeSat deployments. Coelho *et al.* [18], Coelho [19], Ivanov and Bliudze [20], Gonzalez *et al.* [21], Araguz *et al.* [22] have also contributed to this line.

Coelho *et al.* [18], Coelho [19] present the NANOSat MO Framework, which is a standard onboard software framework for nanosatellites that has been 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 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 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 [20] propose a rigorous and robust way to design software. They present the BIP framework, a component-based language that can be used to develop correct-by-construction applications. BIP allows to formally model complex systems and provides a toolset for their 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 reliability, modularity, and portability of the overall system. The CubETH mission is based on four main scientific objectives and used a miniaturized low-power command and data handling 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.* [21] 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 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 through the design of 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.

Araguz *et al.* [22] 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 assessing them qualitatively is, mostly, 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 [23]. The cFS has a solid flight inheritance from NASA projects, and it has also been used in nanosatellites. The cFS provides a unit test suit, but the community has provided SILS interfaces using Simulink and the NOS3 spacecraft simulator [23], [24].

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. They also report 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.

The testing systems applied for the flight software of CubeSats are not deeply 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 the flight software development. In this work, we propose and analyze 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 *et al.* ([25] 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 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. There is still a manual selection after the candidates are generated to assure consistency on tests. 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. This 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.

III. THE SUCHAI FLIGHT SOFTWARE

A. SUCHAI CUBESATS

SUCHAI is a CubeSat based space program that includes the SUCHAI I, II, III, and PlantSat nanosatellites. These satellites are developed by students, engineers, and researchers from different areas 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 [26], [27]. 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, a software solution developed for CubeSat nanosatellites designed to be highly modular and extensible. This flight software is based on the ability to execute generic commands. These commands can be executed automatically from certain modules of the software itself, or they can be sent from the ground station as described in Figure 1. In previous work, Gonzalez *et al.* [10] document the design and implementation of the SUCHAI flight software. In this section, we will describe and explain the most relevant parts of this work.

B. SUCHAI FLIGHT SOFTWARE ARCHITECTURE ADVANTAGES

The SUCHAI flight software architecture is based on the command design pattern adapted for implementation in the C programming language. Figure 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 commands are requested by the *client* modules and derived 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].

We can remark two advantages of the command pattern architecture. First, the operational requirements are mapped to commands, and commands are mapped to functions. Thus, by testing commands execution, we can examine the software robustness and track the associated high-level mission requirements. And second, the command execution follows a single path, independently of the software interaction method. If we decide to interact using the serial console, the communications interface, or a new dedicate *client*, we are testing the complete command execution mechanism, which benefits the test coverage. Therefore, thanks to the implemented architecture, we can integrate different testing techniques into the SUCHAI flight software with minimal code instrumentation.

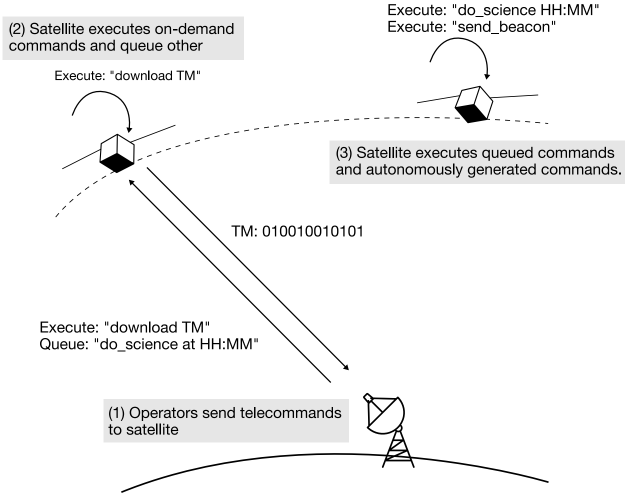

(2) Satellite executes on-demand commands and queue other
Execute: "download TM"
Execute: "do_science HH:MM"
Execute: "send_beacon"
(3) Satellite executes queued commands and autonomously generated commands.
TM: 010010010101
Execute: "download TM"
Queue: "do_science at HH:MM"
(1) Operators send telecommands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

FIGURE 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.

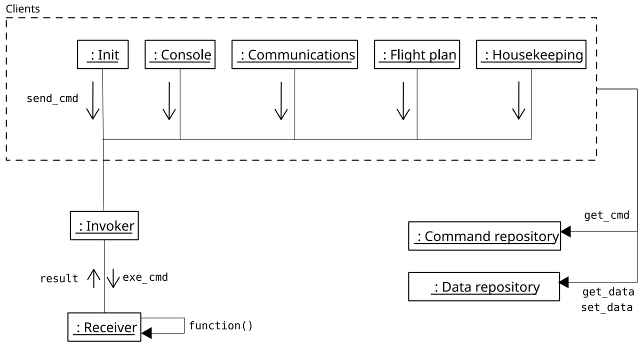


FIGURE 2. The SUCHAI flight software architecture. Adapted from "An architecture-tracking approach to evaluate a modular and extensible flight software for CubeSat Nanosatellites" by C. Gonzalez, C. Rojas, A. Bergel, and M. Diaz, vol 7, pp. 126415, 2019.

C. 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 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, sending the commands under test with fixed parameters, thus covering only particular use cases. In the case of HILS testing, the software is being 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.

Software engineering tools are used on the validation methodology of the SUCHAI flight software architecture. 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.

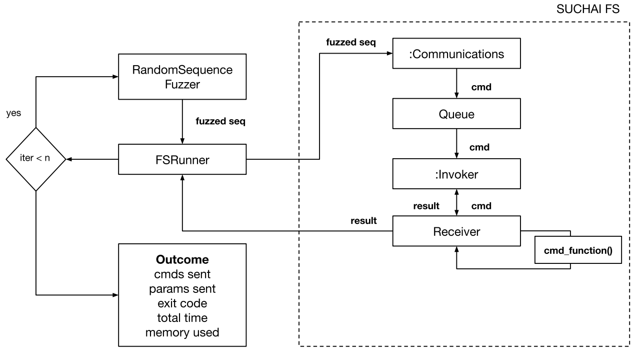


FIGURE 3. Logic diagram of the proposed fuzz testing implementation and the communication system with the SUCHAI flight software. RandomSequenceFuzzer is the system to generate the random sequence of commands to be sent to FsRunner. FsRunner interacts with the SUCHAI flight software running process. 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.

IV. FUZZ TESTING

Fuzz testing is an automated software testing technique that consists in feeding 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 [28].

Section IV-A describes how fuzzing was applied to find unexpected failures on the SUCHAI flight software. The complexity of this application is determined by the SUCHAI flight software architecture. However, it must be emphasized that nothing prevents our approach from being applied to different flight software, as we explain later. Section IV-B lists the different strategies we have employed. Section IV-C presents some aspects when we ran our experiment.

A. FUZZ TESTING ON SUCHAI

As we explained in Section I, the SUCHAI flight software is considered a critical embedded system because it carries out the whole system control procedures of the nanosatellite. Therefore, we are interested in finding vulnerabilities associated with the system's availability and reliability.

There are many ways to apply fuzz testing on the SUCHAI flight software, such as sending random input to functions, modules, or commands. We chose to use this technique with commands because we can take advantage of the software architecture. As we explained above, the SUCHAI flight software architecture is based on the command design pattern, which means that all 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 task of the application. These interfaces will be used to interact with the SUCHAI flight software running process during the execution of the tests. On each test, we will analyze the result of sending a combination of random commands with a random number of parameters and/or random values of parameters. Thus, each test case should be composed of a sequence of commands.

We used the fuzzing architecture proposed in *The Fuzzing Book* Zeller *et al.* [29], which provides a Runner and Fuzzer classes. As described in Figure 3, the Runner represents the process to be executed with the randomly generated data, and the Fuzzer represents the system that generates and feeds this data into a consumer. In this context, FSRunner is a class that inherits from Runner and interacts with the SUCHAI flight software. FSRunner has methods that run this process with the fuzzed commands and parameters. The RandomSequenceFuzzer class inherits from Fuzzer and has methods to generate a sequence of random commands and parameters. We are using the communication system interface to interact with the SUCHAI Flight Software. This interface uses the CubeSat Space Protocol (CSP), which provides a GNU/Linux and FreeRTOS implementation. The GNU/Linux implementation relies on the ZeroMQ library, so we can send commands using ZeroMQ sockets and the local loop.

B. STRATEGIES

The implementation of fuzz testing for the SUCHAI flight software 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:

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 can be achieved by providing sequences of random names of commands without parameters. Thus, the implemented Fuzzer creates N random names of commands. These random names are stored in a list, which is sent to the implemented FSRunner class.

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 exists in the *command repository*, iterating over the list of all the registered commands. If there is not a name in that list that matches with the name of the sent command, the command is not directed to the *invoker* for its execution.

Strategy 1: Random number of parameters. By providing sequences of known commands with a random number of parameters, including zero, this strategy mainly searches for possible errors in the implementations of commands that are not considering the number of the passed parameters. Each parameter is a random value of a fixed type. The types are defined in the command implementation. These are `int`, `long`, `unsigned int`, `float`, and `string`.

In this case and the following ones, the Fuzzer receives a list of available commands implemented in the SUCHAI flight software and the number of commands per sequence. Commands are randomly chosen from the list of available commands. To date, more than 90 commands have been implemented.

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.

Strategy 3: Random parameter values with defined types of values. With this strategy, we look for errors in implementations of commands that have unchecked properties of values, such as the length of each parameter. To achieve that objective, we provide 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, in this case, the types of the values must correspond with the expected types.

C. EXECUTION

The different strategies were executed by sending sequences of 5, 10, 50, and 100 commands. Each of these sequences with a predefined size was generated 1,610 times for each strategy to find useful test cases. Therefore, in total, there were 25,760 sequences executed on the SUCHAI flight software. Initially, the execution of the 25,760 sequences lasted around 3 days. In a replication of the experiment with the same sequences, the execution lasted 175,872 seconds. This translates into 2 days and 53 minutes of total execution time. The replication of the experiment was carried out to analyze time execution on a different computer system with more processing and storage capacity.

V. RESULTS

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

A. EXPERIMENT EXECUTION RESULTS

Initially, we executed the experiment under the operating system Ubuntu version 18.04. In terms of hardware, we used 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 their execution.

The percentages of the failed sequences on each set for strategies 1, 2, and 3 are shown as bar charts in Figure 4, Figure 5, and Figure 6, 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.

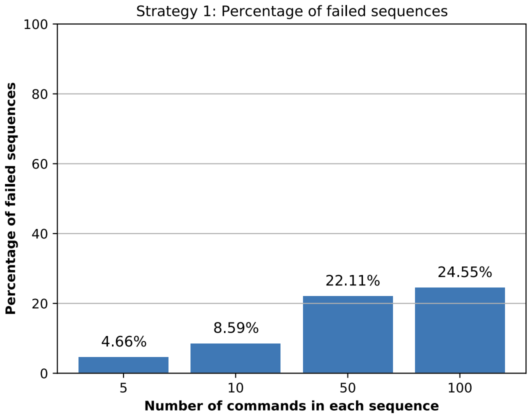


FIGURE 4. Percentage of failed sequences of commands given a fixed number of commands per sequence for strategy 1.

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 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 in total makes up only 0.15% of the sequences. This behavior only appeared in the first execution of the experiment; therefore, it is not particularly related to the experiment performance itself but other factors we will discuss in Section V-B.

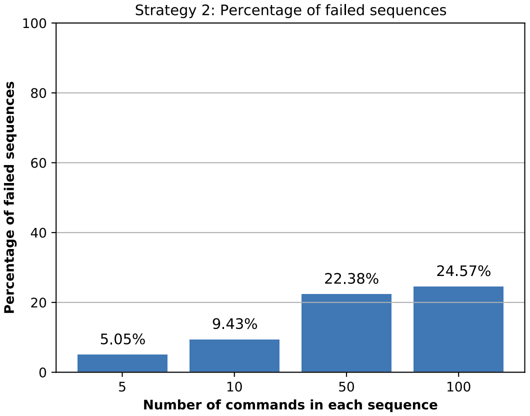


FIGURE 5. Percentage of failed sequences of commands given a fixed number of commands per sequence for strategy 2.

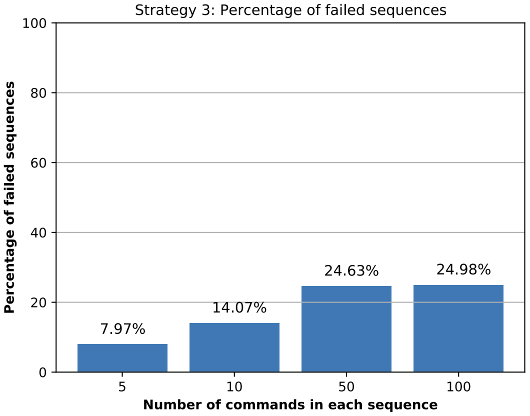


FIGURE 6. Percentage of failed sequences of commands given a fixed number of commands per sequence for strategy 3.

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 7 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 a clearer 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 7 one can identify the commands that made the SUCHAI flight software crash. In fact, 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 V-C.

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

B. EXPERIMENT REPLICATIONS RESULTS

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

In contrast to the first execution of the experiment, we did not observe large differences in the execution time of the sequences. In fact, 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 VI-A.

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 execution of the experiment, we have not found a significant variation of the memory consumption between the sequences. The variation ranges from 11,655 to 12,279 kilobytes.

We found 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. We will discuss this further in Section VI-A.

We conclude that differences in execution time, number of failures, and wrong values in the measurements of memory consumption are not related to an experiment performance issue. This will be further discussed on Section VI-A.

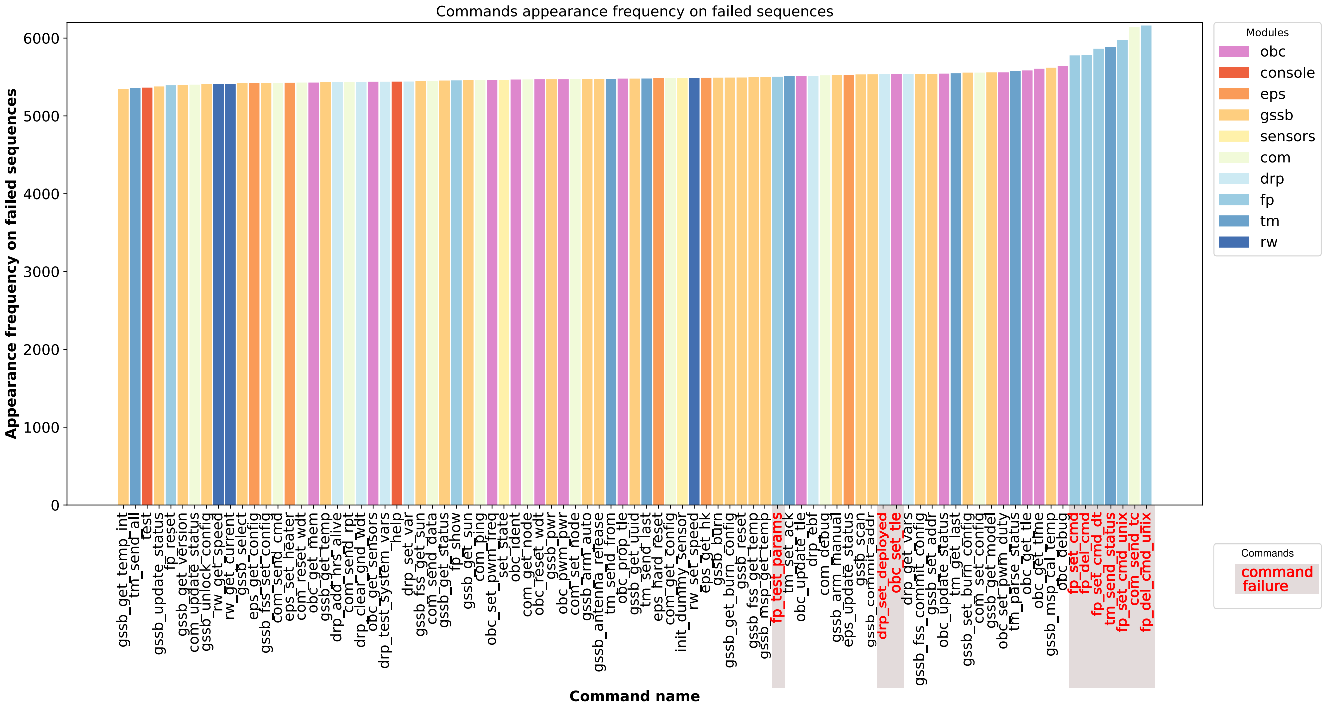


FIGURE 7. **Commands appearance frequency on the failed sequences, classified by command type (module). The commands that made the SUCHAI flight software crash are red-colored in the x-axis labels (identified command failure).**

C. FAILURES FIXING AND CHARACTERIZATION

Once the sequences were sent to the SUCHAI flight software, and the relevant results from their execution were identified, we reported the findings to the software development team. Eight sprint sessions were organized to identify bugs, fix them, and characterize them for a detailed analysis.

1) SPRINTS

At the beginning of each sprint session, the reports made for the software development team were analyzed. This 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 member of the software team tried to identify a failed sequence. When a sequence was found, the issue was reported in 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.[1]

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 to a failed sequence was identified and fixed, three questions were asked to the software team members to better 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’’. We considered the following questions:

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

2) 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 particular command. Ten commands had identified errors. From the questions asked during the sprint sessions, and thanks to the tracking of the code changes, the failure shown in Table 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 that were 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

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

TABLE 1. Characterization of the failures found in the SUCHAI flight software.

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 repoData.c	256	119	10
#5	tm_send_status	-6	FA	■		■	5	2	3	A	A	A	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

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 under study 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 being 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 in Section V, the majority of the software failures we 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 place where the command is being executed. Eight out of the ten presented commands are considered 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.

3) 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. Eight out of twelve bugs were found 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 string without its null character is saved in a buffer. Though the bug is not difficult to find, it required time to understand the cause of 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 bug related to the command *"com_send_tc"* is considered the most difficult to fix since, as we mentioned above, the process to fix it was not direct. The first bug associated with the command *"fp_del_cmd_unix"* was caused by a missing implementation of the functionalities of a certain database system. 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 largest numbers of modified lines of code and modified functions to fix the bug, which are 375 and 10, respectively. This affects its complexity, which was mentioned by the developers beforehand. The bug associated with the command *"tm_send_status"* has 108 modified code lines and 3 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.

4) 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 of them 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.

VI. DISCUSSION

A. THREATS TO VALIDITY

We analyze the threats to the validity for this work as described in *Quasi-experimentation: Design & Analysis Issues for Field Settings* Cook *et al.* [30].

1) 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 original execution. However, the experiment is considered to have low statistical validity because of the low number of executions and the different conditions related to hardware and software characteristics. To mitigate this threat, a higher number of executions is required. Also, the conditions under which the experiment will be reproduced must be defined beforehand. Furthermore, this experiment is considered to have random heterogeneity since 1,610 random sequences for each predefined number of commands per sequence were sent on a particular strategy.

2) INTERNAL VALIDITY

We found very few variations when replicating our experiment. External elements could have affected the executions, possibly attached to the operating system and hardware. In the case of memory consumption and execution time, these results were expected. However, the number of failures also varied: we found fewer failures on each replication than the original execution.

3) 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. For the executions of the experiment, we set up only one standard configuration, considering we were purely testing software. Combinations of values for configuration module variables were not tested. However, several strategies were developed in order to analyze different types of scenarios. Besides, the results considered not only the number of failures but the memory consumption and execution time.

4) 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 run again to find new failures. No new bugs were found.

B. APPLICABILITY TO OTHER MISSIONS

Fuzz testing covers a wide range of strategies, including black-box, white-box, or grey-box testing methods. Particularly, in our work, we implemented fuzzing as a black-box testing method. Then, from our experience in implementing it for the SUCHAI flight software, we describe the basic 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.

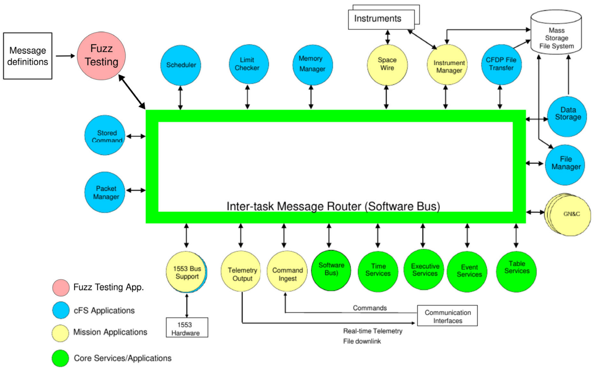


FIGURE 8. cFS top level architecture modified to integrate a fuzz testing application. Adapted from "core flight system (cFS) background and overview", NASA, 2014, https://cfs.gsfc.nasa.gov/cFS-OviewBGSlideDeck-ExportControl-Final.pdf (accessed 2021 June 23).

From the reviewed software architectures by Gonzalez *et al.* [10], the core Flight System [23] and the Command Centric Architecture [11] present a well-documented architecture that fulfills the characteristics previously highlighted. This makes it possible to define a clear way to fuzz both flight software as a black-box testing method.

The core Flight System (cFS) is an open source flight software solution developed by NASA [23]. 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 list of supported messages 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 8 explain this proposal.

The Command Centric Architecture (C2A) is the flight software developed by ISSL researchers at the University of Tokyo with a focus on reconfiguration capability. A major 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 9. 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 the execution of the individual commands sequences or test the spacecraft robustness to deviations in the expected operations.

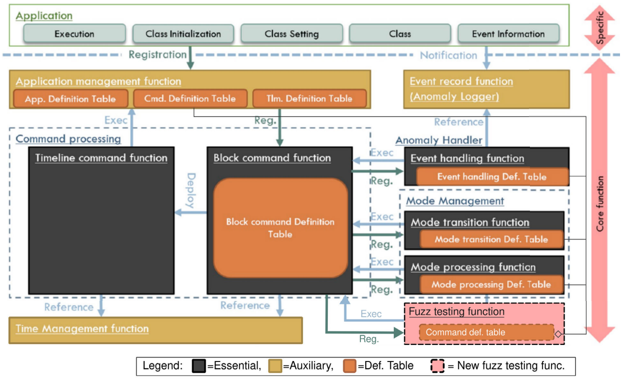


FIGURE 9. C2A software architecture modified to integrate a fuzz testing essential function. Adapted from "Command-centric architecture (C2A): Satellite software architecture with a flexible reconfiguration capability", by Nakajima *et al.*, Acta Astronautica, vol 171, pp. 208-214, 2020.

VII. CONCLUSION AND FUTURE WORK

In this work, we reviewed the flight software testing strategies used in several CubeSat projects and discovered that unit testing, SILS, and HILS are the most common techniques. However, to the best of our knowledge, not all flight software frameworks nor CubeSat missions document the testing procedures used to ensure software quality and robustness. Moreover, in our search of agile testing solutions, we did not find 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, we proposed their use in a context of agile and low-cost CubeSat development, which, to the best of our knowledge, has not been introduced before.

In this work, we explored the usage of fuzz testing techniques in the flight software of the SUCHAI series of nanosatellites by running a set of strategies. We 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 doing more than 1,000,000 commands executions in an unattended manner, we found twelve bugs 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.

The next steps in this research include studying more advanced strategies with a focus on the intelligent identification of failure paths led by code coverage. The proposed fuzz testing application can be extended to other flight software as well. 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 key to achieving the repetition and agility required to test hundreds to thousands of satellites in the context of mega-constellations.

ACKNOWLEDGMENT

The authors would like to thanks the ISCLab and the SPEL team for their commitment and support of this work. We are grateful to Renato Cerro for commenting on an early draft of this paper.

REFERENCES

[1] T. Villela, C. A. Costa, A. M. Brandão, F. T. Bueno, and R. Leonardi, ''Towards the thousandth CubeSat: A statistical overview,'' *Int. J. Aerosp. Eng.*, vol. 2019, pp. 1–13, Jan. 2019.

[2] *Achieving Science With CubeSats: Thinking Inside Box*, National Academies Press, Washington, DC, USA, 2016.

[3] D. L. Dvorak, ''NASA study on flight software complexity,'' in *Proc. AIAA Conf. AIAA Unmanned. Unlimited Conf.*, Apr. 2009, p. 264.

[4] J. Finnigan, ''A scripting framework for automated flight SW testing: Van Allen probes lessons learned,'' in *Proc. IEEE Aerosp. Conf.*, Mar. 2014, pp. 1–10.

[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, no. 12, p. 130, Dec. 2019.

[6] J. Schoolcraft, A. T. Klesh, and T. Werne, ''MarCO: Interplanetary mission development on a CubeSat scale,'' in *Proc. SpaceOps Conf.*, May 2016, pp. 221–231.

[7] J. A. Ledin, ''Hardware-in-the-loop simulation,'' *Embedded Syst. Program.*, vol. 12, pp. 42–62, Feb. 1999.

[8] S. Jeong, Y. Kwak, and W. Jin Lee, ''Software-in-the-loop simulation for early-stage testing of autosar software component,'' in *Proc. 8th Int. Conf. Ubiquitous Future Netw. (ICUFN)*, Jul. 2016, pp. 59–63.

[9] D. J. F. Miranda, M. Ferreira, F. Kucinskis, and D. McComas, ''A comparative survey on flight software frameworks for 'new space' nanosatellite missions,'' *J. Aerosp. Technol. Manage.*, Oct. 2019, pp. 1–13.

[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] C. Boshuizen, J. Mason, P. Klupar, and S. Spanhake, ''Results from the planet labs flock constellation,'' in *Proc. AIAA/USU Conf. Small Satell.*, Aug. 2014, pp. 1–8.

[13] I. F. Akyildiz and A. Kak, ''The Internet of Space Things/CubeSats,'' *IEEE Netw.*, vol. 33, no. 5, pp. 212–218, Sep. 2019.

[14] P. Godefroid, ''Fuzzing: Hack, art, and science,'' *Commun. ACM*, vol. 63, no. 2, pp. 70–76, Jan. 2020.

[15] S. F. Hishmeh, T. J. Doering, and J. E. Lumpp, ''Design of flight software for the KySat CubeSat bus,'' in *Proc. IEEE Aerosp. Conf.*, Mar. 2009, pp. 1–15.

[16] S. Johl, E. G. Lightsey, S. M. Horton, and G. R. Anandayuvaraj, ''A reusable command and data handling system for university CubeSat missions,'' in *Proc. IEEE Aerosp. Conf.*, Mar. 2014, pp. 1–13.

[17] Y. Zaidi, N. G. Fitz-Coy, and R. V. Zyl, ''Rapid, automated, test, verification and validation for the CubeSats,'' *Int. J. Space Sci. Eng.*, vol. 5, no. 3, pp. 242–268, 2019.

[18] C. Coelho, O. Koudelka, and M. Merri, ''NanoSat MO framework: Achieving on-board software portability,'' in *Proc. SpaceOps Conf.*, May 2016, p. 2624.

[19] C. Coelho, ''A software framework for nanosatellites based on CCSDS mission operations services with reference implementation for Esa's OPS-SAT mission,'' Ph.D. dissertation, Inst. Commun. Netw. Satell. Commun., Graz Univ. Technol., Graz, Austria, 2017.

[20] A. B. Ivanov and S. Bliudze, ''Robust software development for university-built satellites,'' 2020, *arXiv:2010.02208*. [Online]. Available: http://arxiv.org/abs/2010.02208

[21] 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,'' *J. Aerosp. Technol. Manage.*, vol. 8, no. 2, pp. 211–231, May 2016.

[22] C. Araguz, M. Marí, E. Bou-Balust, E. Alarcon, and D. Selva, ''Design guidelines for general-purpose payload-oriented nanosatellite software architectures,'' *J. Aerosp. Inf. Syst.*, vol. 15, no. 3, pp. 107–119, Mar. 2018.

[23] D. McComas, J. Wilmot, and A. Cudmore, ''The core flight system (CFS) community: Providing low cost solutions for small spacecraft,'' in *Proc. AIAA/USU Conf. Small Satell.*, Aug. 2016, pp. 1–8.

[24] M. D. Grubb, ''Increasing the reliability of software systems on small satellites using software-based simulation of the embedded system,'' M.S. thesis, Lane Dept. Comput. Sci. Elect. Eng., West Virginia Univ., Morgantown, WV, USA, 2021.

[25] 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 *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Aug. 2019, pp. 975–985.

[26] M. A. Diaz, J. C. Zagal, C. Falcon, M. Stepanova, J. A. Valdivia, M. Martinez-Ledesma, J. Diaz-Peña, F. R. Jaramillo, N. Romanova, E. Pacheco, M. Milla, M. Orchard, J. Silva, and F. P. Mena, ''New opportunities offered by CubeSats for space research in Latin America: The SUCHAI project case,'' *Adv. Space Res.*, vol. 58, no. 10, pp. 2134–2147, 2016.

[27] 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 *Proc. AIAA/USU Conf. Small Satell.*, Aug. 2018, pp. 1–10.

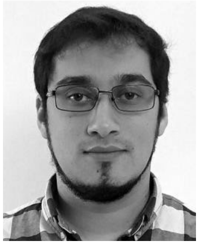
[28] I. Sommerville, *Software Engineering*. Reading, MA, USA: Addison-Wesley, 2011.

[29] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, ''The fuzzing book,'' in *The Fuzzing Book*. Saarbrücken, Germany: Saarland University, 2019.

[30] T. D. Cook, D. T. Campbell, and A. Day, *Quasi-Experimentation: Design & Analysis Issues for Field Settings*, vol. 351. Boston, MA, USA: Houghton Mifflin, 1979.

TAMARA GUTIERREZ received the B.S. degree in computer science from the University of Chile, Santiago, Chile, in 2020, where she is currently pursuing the M.S. degree in computer science. Since 2018, she has been working with the Space and Planetary Exploration Laboratory (SPEL), University of Chile, developing the flight software of the SUCHAI series of nanosatellites. Her work focused on the testing and quality assurance of the flight software.

ALEXANDRE BERGEL received the Ph.D. degree in computer science from the University of Bern, in 2005. Since 2009, he has been an Associate Professor and a Researcher with the University of Chile. He and his collaborators carry out research in software engineering. His effort is about designing tools and methodologies to improve the overall performance and internal quality of software systems by employing profiling, visualization, and artificial intelligence techniques. He has a strong interest in applying his research results to the industry. Several of his research prototypes have been turned into products and adopted by major companies in the semiconductor industry and certification of critical software systems. He authored the book *Agile Visualization, Agile Artificial Intelligence* and coauthored the book *Deep Into Pharo*.



CARLOS E. GONZALEZ received the B.S. degree in electrical engineering from the University of Chile, Santiago, Chile, in 2014, where he is currently pursuing the Ph.D. degree in electrical engineering. From 2011 to 2014, he worked with the Space and Planetary Exploration Laboratory (SPEL), University of Chile, developing the first nanosatellite of the country. His work focused on the development of the flight software and communication system of the SUCHAI satellite. From 2014 to 2016, he worked as a Research Engineer with the Advanced Mining Technology Center (ATMC), University of Chile, developing software for geostatistical applications in the mining industry. Since 2015, he has been a Lecturer of the Computer Networks and Computer Organization courses with the University of Santiago of Chile (USACH).

CAMILO J. ROJAS received the B.S. degree in electrical engineering and the M.S. degree in computer science from the University of Chile, in 2012 and 2020, respectively. From 2011 to 2012, he worked in the first Chilean nanosatellite project, the SUCHAI project, being a developer in the Communication Group. From 2012 to 2016, he worked at Synopsys Chile as a Software Developer with the TCAD Group. After 2016, he returned to the University of Chile and started working as a Researcher with the Advanced Laboratory for Geostatistical Supercomputing (ALGES Laboratory), the Space and Planetary Exploration Laboratory (SPEL Laboratory), where he is currently developing the flight software to be programmed in future satellite missions, and researching about remote sensing techniques to be applied in space environments.

MARCOS A. DIAZ (Member, IEEE) received the degree in electrical engineering from the University of Chile, Santiago, Chile, in 2001, and the M.S. and Ph.D. degrees in electrical engineering from Boston University, USA, in 2004 and 2009, respectively. He is currently an Assistant Professor with the Electrical Engineering Department, University of Chile. His research interests include related to the study of ionospheric turbulent plasma, incoherent scatter radar techniques, low-frequency-radio-astronomy/space instrumentation, and nano-satellite technologies. He is responsible for the Space and Planetary Exploration Laboratory, the Multidisciplinary Laboratory, Faculty of Physical and Mathematical Sciences, University of Chile, where the nanosatellite-based space program at the university is being developed.