1 2 9 0

UNIVERSIDADE Đ
COIMBRA

David Silva de Paiva

FAULT INJECTOR TO VERIFY AND VALIDATE NANOSATELLITES

Dissertation in the context of the Master in Informatics Engineering, Specialization in Software Engineering, advised by Professor Doctor Henrique Santos do Carmo Madeira and presented to Faculty of Sciences and Technology / Department of Informatics Engineering.

July of 2022

1 2 9 0

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE Ɖ
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

David Silva de Paiva

FAULT INJECTOR TO VERIFY AND VALIDATE NANOSATELLITES

Dissertation in the context of the Master in Informatics Engineering, Specialization in Software Engineering, advised by Professor Doctor Henrique Santos do Carmo Madeira and presented to Faculty of Sciences and Technology / Department of Informatics Engineering.

July of 2022

Resumo

CubeSats são pequenos satélites construídos com até 12 unidades na forma de um cubo de 10cm de borda e peso máximo de 10kg e representam uma tendência emergente na indústria espacial. Estes satélites são feitos com componentes *comercial off-the-shelf* (COTS) para reduzir custos e aproveitar a boa relação desempenho/consumo de energia superior dos COTS, que é bastante melhor do que a dos componentes equivalentes de grau espacial, que são concebidos para suportar radiação. Infelizmente, os componentes COTS são suscetíveis a *Single Event Upsets* (SEU), que são erros transitórios causados pela radiação espacial. Os SEU tornam o estudo do impacto de falhas causadas por radiação espacial uma etapa obrigatória nas fases de Verificação e Validação (V&V) do desenvolvimento de software para CubeSats, a fim de avaliar cuidadosamente os pontos fracos que devem ser reforçados através do uso de técnicas específicas de tolerância a falhas de software. O facto do impacto das falhas ser fortemente dependente do software executado no hardware COTS sugere que o estudo do impacto das falhas de radiação deve ser realizado sempre que o software do CubeSat sofrer uma grande alteração, ou até mesmo uma pequena atualização.

Esta tese apresenta o CubeSatFI, uma plataforma de injeção de falhas para CubeSats destinada a facilitar a incorporação desta etapa extra no software de Verificação e Validação do CubeSats. CubeSatFI permite a fácil definição de campanhas de injeção de falhas que emulam os efeitos da radiação espacial. SEU são emulados de forma realista através de falhas de bit-flip injetadas nos registos do processador e noutros locais das placas CubeSat que podem ser alcançadas por *boundary-scan*, que está disponível nas placas CubeSat através da porta de acesso de teste JTAG. A execução das campanhas de injeção de falhas é controlada pela plataforma CubeSatFI de forma totalmente automatizada.

A eficácia do CubeSatFI é demonstrada com o EDC (Environment Data Collection), uma *payload board* que será usado numa constelação de satélites do Instituto Nacional de Pesquisas Espaciais Brazileiro (INPE), fornecendo uma visão realista sobre o impacto de falhas no software EDC.

Palavras-Chave

CubeSats, Componentes COTS, Injeção de falhas, Verificação e Valdidação, Erros Transitórios

Abstract

CubeSats are small satellites built with up to 12 units of the shape of a cube of 10cm edge and weight of 10kg maximum and represent an emergent trend in the space industry. These satellites use commercial off-the-shelf (COTS) components to reduce cost and take advantage of the superior performance/power consumption ratio of COTS, which is an order of magnitude better than the equivalent radiation-hardened space-grade-components. Unfortunately, COTS components are susceptible to Single Event Upsets (SEU), which are transient errors caused by space radiation. SEU makes the study of the impact of faults caused by space radiation a mandatory step in the development of CubeSats software, in order to carefully evaluate weak points that must be strengthened through the use of specific software fault tolerance techniques. The fact that the impact of faults is strongly dependent on the software running on the COTS hardware indicates that the study of the impact of radiation faults must be carried out every time the CubeSat software has a major change, or even a minor update.

This thesis presents CubeSatFI, a fault injection platform for CubeSats meant to facilitate the incorporation of this extra step in the Verification and Validation of CubeSats software. CubeSatFI allows the easy definition of fault injection campaigns that emulate the effects of space radiation. SEU are emulated realistically through bit-flip faults injected in the processor registers and in other locations of the CubeSat boards that can be reached by boundary-scan, which is available in CubeSat boards through JTAG Test Access Port. The execution of the fault injection campaigns is controlled by the CubeSatFI platform in a fully automated mode.

The effectiveness of CubeSatFI is demonstrated with the EDC (Environment Data Collection), a payload system that will be used in a constellation of satellites from the Brazilian National Institute for Space Research (Instituto Nacional de Pesquisas Espaciais - INPE), providing a realistic insight on the impact of faults in the EDC software.

Keywords

CubeSats, COTS components, Fault Injection, Verification and Validation, Transient Errors

Acknowledgments

First, I want to thank my parents Fernanda and Amadeu for all the efforts and sacrifices they have made and continue to make so that I can achieve my goals successfully. None of this was possible without them. And also, I want to thank my brother Moisés.

I thank the University of Coimbra, in particular the Department of Informatics Engineering, and the teachers I came across for all the knowledge that they transmitted to me.

I thank the Center for Informatics and Systems of the University of Coimbra (CISUC) for the opportunity to carry out the research work presented in this thesis. Special thanks to my lab colleagues Anamta, Qianying, and Omid. What a wonderful journey by your side.

I thank National Institute for Space Research of Brazil (INPE) for the opportunity to test the tool developed on a real satellite.

I thank Professor Henrique Madeira, from the University of Coimbra, for all the monitoring, dedication, and words of strength and encouragement. Special thanks for all the support.

I also want to thank the CONASAT team from INPE for welcoming me and supporting my journey in the last year, especially Doctor José Marcelo and Engineer Raffael Staditte.

I would like to thank my friends for every moment, every night of work during the last five years, but also for the partying and funny moments. Thanks for all the friendship, especially during this phase. Special thanks to my dear friends Sara, Margarida, Carina, Rafael, Francisco, Simão, Tiago, Simão Jr., and Haitham.

Contents

List of Abbreviations

ADVANCE	Addressing Verification and Validation Challenges in Future Cyber-Physical Systems
ALU	Arithmetic Logic Unit
CDS	CubeSat Design Specification
COTS	Commercial off-the-shelf
CPS	Cyber-Physical Systems
DDC	Data Distribution Center
EDC	Environment Data Collector
GPR	General Purpose Registers
GP	Ground Platforms
JTAG	Joint Test Action Group
ICAP	Internal Configuration Access Port
INPE	Brazilian National Institute for Space Research (Instituto Nacional de Pesquisas Espaciais Brazileiro)
IECU	Instruction Execution Control Unit
LEO	Low Earth Orbit
MMU	Memory Management Unit
OBC	On-Board Computer
OpenOCD	Open On-Chip Debugger
SDC	Silent Data Corruption
SEU	Single Event Upsets
SoC FPGA	System on Chip - Field Programmable Gate Array
SWIFI	Software Implemented Fault Injection
SWIFT	Software Implemented Fault Tolerance
TAP	Test Access Port
V&V	Verification and Validation
VALU3S	Verification and Validation of Automated Systems' Safety and Security

List of Figures

List of Tables

Chapter 1
Introduction

Nowadays, the interest in the development and deployment of CubeSats solutions has become a trend in the space industry. CubeSats are small satellites built with up to 12 units in the shape of a cube of 10cm edge and weight of 10kg maximum, according to the CubeSat Design Specification (CDS) – a standard (de facto) for mechanical design and interfacing for satellites [1].

This thesis presents CubeSatFI, a fault injector platform that aims to help space agencies and software developer teams with an effective tool to verify and validate CubeSats software.

This chapter introduces the scope and motivation of this thesis and provides an overview of the structure of the entire document.

1.1 Context of the project

This thesis was developed in the context of the European H2020 ADVANCE ("Addressing Verification and Validation Challenges in Future Cyber-Physical Systems"). The scientific objective of the ADVANCE project is to conceive new approaches to support the Verification and Validation (V&V) of Cyber-Physical Systems (CPS). It will explore techniques, methods, and tools applicable to different phases of the system lifecycle, but always with the final objective of improving the effectiveness and efficacy of the V&V process [1].

Within the ADVANCE project, a specific partnership was carried out with the Brazilian National Institute for Space Research (Instituto Nacional de Pesquisas Espaciais - INPE), which is one of the project partners. In short, INPE aims to empower Brazil in scientific research and space technologies and is an international reference in research in space and atmospheric sciences, space engineering, meteorology, and Earth observation using satellite images and studies of climate change[2]. Hence, INPE proposed the development of a tool capable of exhaustively testing the effect of space radiation on their CubeSats satellites, particularly the effects of space radiation on the behavior of the software running on the CubeSats boards. Since CubeSats use commercial off-the-shelf (COTS) components to reduce cost and to seize the superior performance/power consumption ratio of COTS when compared to traditional radiation-hardened electronics, the challenge is to emulate the hardware faults caused by space radiation with the least perturbation possible to the rest of the CubeSat board (target system), and without compromising the development costs.

[1] https://cordis.europa.eu/project/id/823788 (accessed Jan. 20, 2022)
[2] http://inpe.br/faq/index.php?pai=1 (accessed Jan. 13, 2022)

Moreover, this thesis was also developed under the VALU3S ("Verification and Validation of Automated Systems' Safety and Security") project. The VALU3S project aims to evaluate the state-of-the-art V&V methods and tools, and design a multi-domain framework to create a clear structure around the components and elements needed to conduct the V&V process[3].

The work developed under this thesis was supported in part by the Grant CISUC-UID/CEC/00326/2020, funded in part by the European Social Fund, through the Regional Operational Program Centro 2020, and supported in part by the H2020, Marie-Curie project ADVANCE ("Addressing Verification and Validation Challenges in Future Cyber-Physical Systems"), project funded by the VALU3S ("Verification and Validation of Automated Systems' Safety and Security") project, supported by the European Commission, and in part by European Leadership (ECSEL) Joint Undertaking (JU) under Grant 876852, and in part by the JU from the European Union's Horizon 2020 Research and Innovation Programme.

1.2 Motivation

CubeSats are made of commercial off-the-shelf (COTS) components (both hardware and software). The use of COTS-based systems in mission-critical applications is an established trend in industry sectors. They offer a real opportunity to reduce development costs and deployment times, which greatly explains the growing interest in using COTS components in mission-critical systems. Additionally, COTS components normally benefit from a large installation base in a multitude of configurations, which is often considered as an effective "test in the field".

However, COTS components are not prepared to deal with the demanding space conditions. In space, high-energy ionizing particles exist as part of the environment and these particles, often generically called as space radiation, may cause problems in the electronic circuits of the satellites. This is a well-known fact, well documented by the scientific community [2]–[6], and is something that space agencies need to have in mind when it is time to develop and deploy a satellite made of COTS components.

The sensitivity of COTS components to space radiation is the main reason why CubeSats (and other miniature types of satellites) have been regarded as not adequate for high-priority and critical missions due to its low reliability [7]. However, the situation is changing very quickly and the dramatic advantages of CubeSats in terms of cost, weight, energy and easiness of development and deployment have changed space agencies plans and private investors, which currently look at CubeSats as a concrete solution for a wide variety of missions, including critical missions.

The challenge is on how to develop reliable CubeSats, capable of tolerating the stringent requirements of space missions, despite being made of low cost and radiation sensitive components. Is it possible to make CubeSats software resistant to space radiation? To

[3] https://valu3s.eu/ (accessed Jan. 20, 2022)

help to answer this question, INPE proposed the development of a platform capable of assessing the behavior of CubeSat software in the presence of radiation. Furthermore, INPE made available the Environment Data Collector (EDC), a CubeSat payload board for the Brazilian Environmental Data Collection System that will be used in all the nanosatellites from the CONASAT project [8]. In this thesis, the EDC payload system will be used as a realistic example to design and test the fault injection tool and propose a pragmatic approach to assure that CubeSats software will perform in a reliable manner in the presence of space radiation.

1.3 Objectives

Taking into account the need to build more reliable CubeSats, this thesis has three main objectives that are presented in the following paragraphs.

The **first objective** is concerned with the development of a tool – CubeSatFI - capable of reproducing the effects of radiation-induced faults in CubeSats boards. Fault injection is an "old recipe" that can be applied in this new context created by the CubeSats boom. The goal is not to use this tool to evaluate satellites or their hardware, as it is typically done by previous fault injection studies. In contrast, this tool aims to be an important instrument to evaluate the impact of radiation-induced faults on the software that runs on top of COTS-based CubeSats. A known fact from fault injection studies is that the impact of transient faults in computer systems is highly dependent on the concrete software code [9]. Furthermore, this tool intends to be used both in the evaluation of the impact of transient faults on the software integrity and behavior, as well as in the evaluation of possible Software-Implemented Fault Tolerance (SWIFT) techniques that will be subsequently used to make CubeSats software resistant to space radiation.

As already mentioned, COTS components are not prepared to deal with space radiation. With that in mind, the **second objective** of this thesis is to prove that the negative impact caused by space radiation can be mitigated or even tolerated by software-implemented fault tolerance techniques. SWIFT techniques are known and well documented in the literature, and the author considers that the use of SWIFT techniques is the right solution to increase the reliability of CubeSats software without degrading the important advantages of the COTS-based approach used by these satellites, particularly the low costs, low weight, and low energy consumption of CubeSats. The CubeSatFI tool will be used to evaluate the effectiveness of SWIFT techniques in the EDC payload board.

And, last but not least, this thesis aims to formalize the use of fault injection in the software development process for CubeSats, particularly in the Verification and Validation phases. Although fault injection is a widely used technique in several industrial application areas, including in the space domain, the concrete application of fault injection in the CubeSat industry requires a new perspective and leads to new ways of using fault injection in the development of CubeSats and, more specifically, in the software verification and validation phases. As already mentioned, the impact of transient hardware faults in computer systems is highly dependent on the actual code running on such systems. When the code changes, the impact of faults could change drastically. In the case of CubeSats development, this means that the evaluation of the impact of radiation-induced faults must be carried out every time the CubeSat software changes, even when such changes are just a minor update. In other words, this thesis proposes that fault injection must be included as a mandatory step in the development of software for CubeSats.

1.4 Tangible Contributions

During the development of the thesis, some key contributions were achieved. The following paragraphs summarize these contributions:

- CubeSatFI: A fault injection tool to emulate faults caused by single event upsets in the processor registers of CubeSat boards.
- A scientific paper [10] that presents the CubeSatFI platform and a first real example of its usage was submitted to the 2021 Latin-American Symposium on Dependable Computing (LADC) and published in the conference proceedings by the Institute of Electrical and Electronics Engineers (IEEE) at IEEE Xplore. The paper can also be found in Appendix B - Fault injection platform for affordable verification and validation of CubeSats software.
- The proposal of an enhanced software development process for CubeSats to cope with space radiation-induced faults. This proposal will be presented in a scientific paper that will be submitted to the 27th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2022). The current version of the paper can be found in Appendix C - Enhanced software development process for CubeSats to cope with space radiation faults.

1.5 Thesis Structure

In addition to this introduction, the thesis is organized in different chapters that are briefly described in the following paragraphs.

Chapter 2: Background and State of the Art aim to introduce basic concepts related to small satellites and CubeSats. Furthermore, chapter 2 presents relevant related work on fault injection for space applications, compares existing fault injection tools for similar target environments, and presents a brief explanation on software fault tolerance techniques.

Chapter 3: CubeSatFI Requirements and Architecture presents the CubeSatFI requirements and architecture.

Chapter 4: CubeSatFI Functional View aims to present the fault injector and demonstrate the effectiveness of the tool on the Environment Data Collector (EDC) CubeSat board that will be used on a real space mission.

Chapter 5: Integration of Fault Injection in the Software Development Process proposes the enhancement of the software development process for CubeSats to cope with the space radiation-induced faults.

Chapter 6: SBCDA Use Case and Results demonstrate the proposed integration of fault injection in the software development process for CubeSats using three different software applications running in the EDC.

Chapter 7: Conclusions and Future Work concludes the thesis, presents the main outcomes and results, and highlights possible future work directions.

Chapter 2
Background and State of the Art

This chapter provides basic background concepts on small satellites and CubeSats. In addition, it presents the state of the art on fault injection for space applications with a brief presentation of the most relevant fault injectors and fault tolerance techniques that can be used to develop more reliable software for CubeSats.

2.1 Small satellites and CubeSats

There is a wide variety of small satellite types, which has been almost exclusively used in low Earth orbits for applications such as Earth observation and remote sensing or communications. The most common type among small satellites is known as nanosatellite, which includes satellites with a mass of up to 10 kg. Since 2012, the number of launches of nanosatellites has grown significantly as shown in Figure 1.

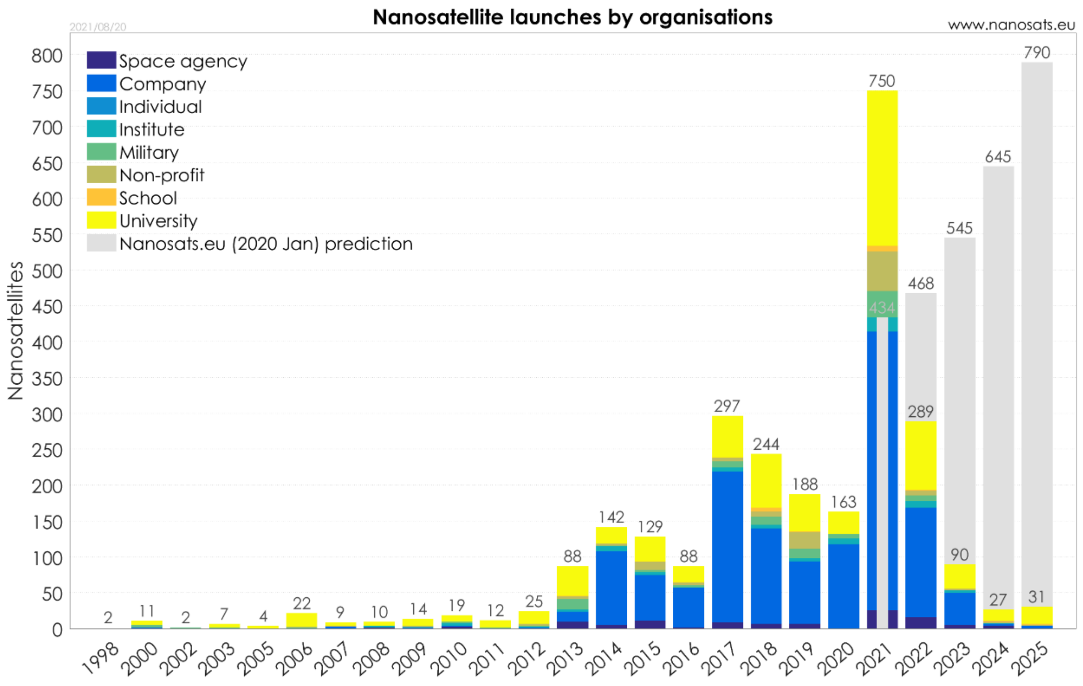


Figure 1 - Number of nanosatellites launches per year [4]

This growth in the number of launches is largely due to the popularization of CubeSats. These satellites follow the CubeSat Design Specification (CDS), which is a standard (de facto) for mechanical design and interfacing for satellites [1]. CubeSats strongly reduce

[4] https://www.nanosats.eu/ (accessed Oct. 29, 2021)

the cost and development time of space projects, increase accessibility to space, and allow sustained frequent launches. This standard defined the 1U format, a 10cm cube edge for the satellites, and other formats derived from it, 1.5U, 2U, 3U, 6U, etc. – shown in Figure 2. Satellite subsystems such as solar panels, antennas, on-board computer, power system, communication systems, and others started being sold as COTS, and the standardization of the interface between the satellite and the launcher also simplified the provision of the launching service.

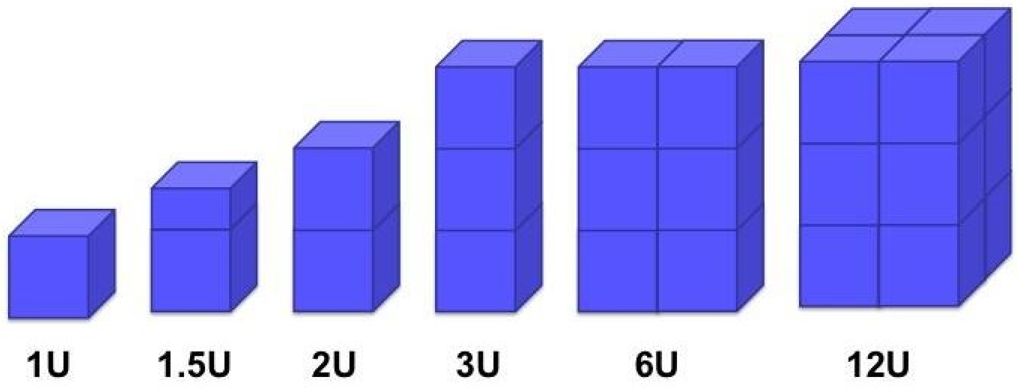


Figure 2 - Common CubeSats Configurations [5]

CubeSats are generally launched into Low Earth Orbits (LEO), at an altitude of up to 600 km. This is mainly due to the small size that limits the energy available in the power supply and makes it impossible to use high-gain antennas. Launching CubeSats into higher altitude orbits, such as the geostationary orbit around 36,000 km, would need high power transmitters to achieve high transmission rates, which is difficult to accommodate in the CDS standard.

Among the popular CubeSat applications, stand out communication services such as IoT and Space Internet, remote sensing with the acquisition of images of the Earth and space, and geolocation.

The CubeSats manufacturing takes special advantage of the use of COTS components because COTS largely outperforms (in performance, cost, weight, etc.) components that are qualified for space applications, which open opportunities to develop new space technologies and carry out space missions in the fastest and cheapest way. The spatial qualification process slows down components, as the space industry mainly limits the operating frequency and dynamic use of the processor memory cache to reduce the inherent risks of radiation suffered in space. COTS components also have the advantage

[5] https://www.nasa.gov/content/what-are-smallsats-and-cubesats (accessed Nov. 02, 2021)

of cost and ease of purchase, eliminating potential embargo issues due to the protected nature of many space components.

The use of COTS-based systems in space missions is particularly attractive, especially in the context of CubeSats and nanosatellites where very low cost and very quick development time are paramount goals. However, in spite of the advantages of COTS components (low cost, top performance, low energy consumption, readily available for purchase), the reality is that COTS are not usually designed for the stringent requirements of space missions. In fact, errors caused by SEU are established as the major cause of COTS components failures in space [6]. The impact of space radiation could damage COTS components on a permanent basis, but the most common effect is to cause transient faults [6] that may lead the software to crash or produce erroneous results. This means that the actual use of COTS components in space missions must be preceded by a careful study of the impact of faults caused by space radiation on system behavior. This is a necessary step, even for Low Earth Orbit (LEO) (and low-risk) CubeSat missions.

Hardware COTS components used in boards of CubeSats are sensitive to space radiation. That is a known fact, well documented in the semiconductors data sheets, and abundantly evaluated by researchers and practitioners, using ground radiation and methods described in the standard ISO 21980:2020 [2], and even in on-orbit measurements (e.g., [3], [4]). Space radiation can cause a variety of effects in COTS microelectronics, ranging from permanent failures to transient faults, depending on the different sources of space radiation and on the radiation exposure [5]. However, transient faults caused by SEU are recognized as the major cause of component malfunctions in space [6], especially in the LEO used by CubeSats. In other words, the major risk resulting from space radiation in CubeSats is the increased rate of hardware transient faults that may cause erroneous behavior in the software running on CubeSat boards.

Obviously, CubeSats boards can be designed to reduce the probability of hardware transient faults due to SEU. For example, using better COTS components (often called COTS+ [11]) and/or including some hardware fault tolerance mechanisms in the design of the boards. But the use of full-fledged hardware fault tolerance (e.g. TMR [12]) would be prohibitive in terms of power consumption and weight, and in practice CubeSats boards only have lightweight mechanisms such as memory error detection and correction. CubeSat proposals with strong fault tolerance mechanisms are rare, but even the few ones available, such as a recent proposal presented in [13], rely partially on software fault tolerance techniques [14] (with some support from the COTS hardware architecture in case of [13]).

Although CubeSats generally use ordinary hardware COTS components (i.e., components sensitive to space radiation), typical architectures of CubeSats boards [15] include several mechanisms to cope with SEU and faults caused by space radiation. Memory is typically protected through error detection and correction codes, and communication structures also use error detection and correction provided by the communication protocols and associated hardware of the communication links. Memory, in particular, represents a large silicon surface exposed to radiation, which means that protecting memory from transient bit-flip errors due to space radiation is mandatory.

Fortunately, the protection of memory and communication channels against transient faults caused by space radiation is relatively easy to achieve at low cost because of the regular nature of such structures. For example, the use of extended Hamming codes [16] to assure single error correction and double error detection in the memory just requires two extra parity bits and is a frequent solution in CubeSat boards. Similarly, the use of communication protocols and techniques such as forward error correction codes [17] are effective in dealing with errors caused by transient faults in communication channels.

The big challenge is to protect the processor(s) of CubeSat boards from the effects of space radiation. Obviously, the use of space-grade processors that resist space radiation is not an option for CubeSats, as the cost of such processors is several orders of magnitude higher than the cost of common COTS processors. But, unfortunately, COTS processors are not immune to space radiation and, at the same time, the complex internal structure of processors does not allow the use of affordable data error detection and correction methods that protect uniform and regular structures such as memories and communication channels. In other words, existing CubeSat boards can deal with transient faults caused by space radiation that affect memory and communication, but the processor represents the major weakness of the reliability of CubeSats.

The obvious solution would be to rely on classic fault-tolerant architectures at the board level [18] to tolerate faults of the COTS processors in CubeSats. But these techniques represent a substantial increase of hardware redundancy, with a high negative impact on the board weight and power consumption. For example, the use of duplicated processors in CubeSat boards would require a large amount of additional hardware to deal with the comparison of the two processors, no matter the concrete flavor of fault-tolerant architecture used in the board design. For example, techniques such as lock-step dual-processor architectures would require the low-level comparison of the hardware signals of both processors (and, most likely, can only be used when the processors are implemented in FPGAs to have access to the internal processor structure to allow synchronization of signals). Other architectures such as symmetric multiprocessors (i.e., two or more identical processors sharing a single main memory) would also need substantial additional hardware and have a negative impact at other levels (e.g., would require a multiprocessor-aware operating system) [18].

Recent research work (Ph.D. thesis of C. Fuchs, December 2019 [19]) proposes a novel on-board-computer architecture for very small satellites (<100kg) capable of achieving high reliability without using radiation-hardened semiconductors, through the combined use of hardware and software-implemented fault tolerance techniques [19]. However, in spite of this promising research result from C. Fuchs, to the best of the author's knowledge, there are no fault-tolerant boards available for CubeSats, especially boards that can cope with transient faults that affect the processor, which are the major threat to the reliability of CubeSats.

The current situation in the space industry is that, in spite of the growing interest in CubeSats, this category of miniature satellites is still considered as not adequate for high-priority and critical missions, and the reason is the low reliability of CubeSats [7]. Data from 178 launched CubeSats show that the 2-year reliability estimation ranges from 65% to 48% [7]. The detailed analysis of the results presented in [7], concerning the subsystem identified as the root cause of the failure, shows that the payload subsystem contributes with modest figures (from 3% to 4%), which make sense in an analysis focused on failures of CubeSat missions with a strong incidence of DOA (dead-on-arrival), where the satellite never achieved a detectable functional state. However, it is possible to speculate that the failure rate in CubeSat payload software could be much higher, especially considering transient failures in the payload software that, apparently, has not been considered in [7].

With the ideas presented above, two significant points emerge and need to be addressed:

- The evaluation of the impact of space radiation in CubeSats should be focused on the impact of processor faults, as the processor (i.e., the chip) is the key element of a CubeSat that cannot be protected or replicated without damaging the main advantages of low cost, low weight, and low energy consumption of CubeSats. SEU-induced hardware transient faults in the processor directly affect the software execution, being visible as potentially erroneous behavior of the CubeSat software with all sort of possible negative consequences.
- The use of software fault tolerance techniques seems the most promising approach to improve CubeSats reliability and resilience in terms of space radiation, while keeping the affordable budget, low energy, low mass, and easy to purchase hardware components of CubeSats.

2.2 Fault Injection for Space Applications

Fault injection consists of "the deliberate insertion of artificial faults in a computer system or component in order to assess its behavior in the presence of faults and allow the characterization of specific dependability measures" of the target computer system [20]. This is a mature technology that has been established as an attractive way of validating specific fault handling mechanisms and as an effective technique to provide experimental data for the estimation of fault-tolerant system measures, such as fault coverage and error detection latency [21], [22].

There is a wide variety of techniques to inject faults in computer systems. First proposals of fault injection techniques used heavy-ion radiation into processor chips [23], pin-level fault injection [24], or electromagnetic interference [25]. However, the increased complexity of computer systems made these techniques nearly impossible to apply, not only because of the inherent difficulties of controlling the injection process in highly complex processors but also because of the difficulties in the collection of high-quality information on the target system behavior after the injection of the faults. In practice, the initial fault injection techniques have been replaced by the emulation of faults through software mechanisms, which is known as SWIFI (Software Implemented Fault Injection).

One of the first SWIFI proposals was [26], which was subsequently replaced by more effective and less intrusive methods such as [20], [21], [23]–[27] that became the standard de facto in fault injection. The key idea of SWIFI tools is to emulate hardware faults through software, using very small interruption response routines, with just a few instructions (or scan-chain debugging resources in [27]) that insert bit-flip errors in very low-level structures (e.g., processor registers, memory, or busses, among others).

Initially, fault injection techniques only emulated (realistically) hardware faults through bit-flip and stuck-at-bit fault models. One of the first studies that investigated the possibility of emulation of software faults (i.e., software bugs) using fault injection tools was published in [28]. The first practical approach to inject realistic software faults (that emulates real bugs found in deployed software) was presented in [29]. A relatively recent survey on software fault injection techniques and tools can be found in [30]. Despite this, the study of software fault injection is out of the scope of this thesis.

Concerning the injection of faults that emulate the effects of SEU, the heavy-ion methods are in fact a direct injection of real radiation-induced faults [23]. But this approach was replaced by SWIFI methods [20] since it is very difficult to apply (it requires a radiation source provided by radioactive material or complex cyclotron facilities) and cannot be used in modern processors in a controlled manner. SWIFI techniques have become the dominant fault injection method used by space agencies such as NASA [9] or ESA [31] for general purpose evaluation of satellite computer systems in faulty conditions.

Consistent results in fault injection studies show that the impact of transient faults on the software behavior (i.e., the failure modes observed) is highly dependent on the actual software under evaluation. In [32], for example, faults injected while the target system was running quite diverse software, selected from many application areas such as automotive, telecomm, office, etc., show differences in the percentages of failure modes observed higher than 70%, depending on the actual software running. In another work, where injected faults were intended to emulate SEU-induced faults in a NASA COTS-based payload for scientific data processing onboard the satellite, the differences observed in the fault impact for the different programs running on the system reached 45% for some failure modes [9].

The fact that the impact of transient faults is highly dependent on the actual software suggests that fault injection should be a mandatory step in the development of CubeSats software to evaluate the sensitivity of the CubeSat software to specific failure modes, especially to failure modes that may affect the CubeSat mission. Even after relatively minor updates in the CubeSat software that may change the software behavior in the presence of SEU-induced faults, it is recommended to perform fault injection tests to (re)validate CubeSat resilience to space radiation. The CubeSatFI platform is intended to facilitate the integration of fault injection as a mandatory step in the verification and validation of CubeSats software.

Concerning fault injection tools available, in spite of the fact that fault injection is a well-known technique that was been applied in a broad range of utilization contexts, the reality is that there are no much fault injection tools readily available. In particular, for the specific context of fault injection in CubeSat boards and CubeSats applications, to the best of the author's knowledge, there are no fault injection tools currently available. That is the main reason why it was decided to develop a new fault injection tool specifically targeted to CubeSat board.

CubeSatFI fault injection platform developed and presented in this thesis **emulates hardware transient faults** in order to evaluate the CubeSat software resilience against space radiation-induced faults. CubeSatFI uses SWIFI techniques inspired by [20] (Xception) and, particularly, it uses the scan-chain method proposed in [27]–[31], [33], [34].

Concerning the Xception [20] approach, it relies on the use of special CPU debugger registers to inject the faults with the minimum perturbation on the target system. However, this approach causes a strong coupling between the fault injection tool and the target system. In addition, Xception was the first fault injection tool that uses the interruption-based approach to inject faults with minimal intrusion, at least considering the intrusion from the point of view of the amount of instructions executed to inject each fault. CubeSatFI has got some inspiration from the interruption-based approach used by Xception, even if the execution context is very different and CubeSatFI does not suffer from the strong coupling effect between the fault injector tool and target system that is one of the problems of Xception [20].

GOOFI [27] is the most prominent tool of the tool family using the scan-chain approach and one of the few tools currently under utilization, at least in research contexts, which makes it a relevant reference for the fault injector tool developed in this thesis.

Scan-chain approaches (see [27], [35]–[38]) has the advantage of using the features available in the target system for testing purposes to inject faults with minimal perturbation of the system (other than the injected fault). Since most of the CubeSat boards include these scan-chain features, the proposed CubeSatFI can be used in most of the CubeSat satellites.

Another variant of fault injection tools includes the ones that intend to test the robustness of Field Programmable Gate Arrays (FPGAs) against space radiation and are briefly presented in [39]. These types of fault injectors are concerned with the evaluation of the hardware behavior. In contrast, CubeSatFI is concerned with the sensitivity evaluation against radiation of the software that runs on top of a CubeSat, making this type of fault injector not relevant to this thesis.

FIRED [40] was proposed to evaluate the dependability of critical systems built on SRAM-based FPGAs through the emulation of hardware faults by injecting bit-flips in the SRAM memory cells through the Internal Configuration Access Port (ICAP). The faults injected by FIRED can produce errors in the design of VHDL (very high-speed integrated circuit hardware description language) or Verilog modules deployed in the FPGA [40]. The nature and purpose of FIRE is quite different from the CubeSatFI, which makes FIRE not particularly relevant for our context.

Some other interesting tools, although not well aligned with the goals of the present thesis, are SEInjector [41] and GRINDER [42]. SEInjector was developed to simulate soft errors in the registers of x86 target systems allowing to perform fault injection in a specific code segment. And GINDER was developed thinking on reusability (i.e., the ability to use the tool on different target systems with the minimum changes possible on the tool source code) and therefore its architecture is focused on high adaptability to accommodate new target systems.

Taking into account the context of this thesis, Xception [20] and GOOFI [27] are the reference tools that have inspired the design and development of CubeSatFI. Xception is the reference for an interrupted-based approach and GOOFI is the most relevant tool using boundary-scan. With that in mind, these fault injectors are briefly presented in the following paragraphs.

Xception

Xception is a fault injection tool and monitoring environment that intends to emulate transient faults in several processor units of the target system [20]. These units can be Instruction Execution Control Unit (IECU), General Purpose Registers (GPR), Memory Management Unit (MMU), among others.

Back to the end of the ’90s, the Xception took advantage of the most advanced debugging and performance monitoring capabilities available in the microcontrollers of that time. Hence, the tool was able of emulating transient faults with the minimum perturbation for the target system. This was accomplished since intrusive techniques like modifying the application source code, adding software traps in the code, or executing the application in debugging mode are not used by the Xception tool to perform the fault injection.

The Xception is composed of several modules which include the main module that makes available the user interface for fault definition, fault execution, and collection of the effect produced by the injected faults and runs on a host computer. In addition, Xception has a module responsible for communicating with the kernel of the target system. And last, a set of functions that should be called to start the fault injection campaign.

The tool makes available a set of fault triggers that can be used to emulate transient faults. It is possible to define a fault that will be injected after a specified time interval thus making the definition of faults through a time trigger. Spatial triggers are also available, i.e. the tool allows to inject a fault when some instruction that is stored in a specific address is fetched or even when some data saved in some address is accessed. This last one allows to perform inject on reading and inject on writing, i.e., inject a fault when some data is written or read from a specific address. Finally, the Xception allows the combination of all these triggers in order to create a diversity fault injection campaign.

The results obtained at the end of each injection are saved in a file in a spreadsheet format. Hence can be analyzed using powerful data analysis tools.

GOOFI : Generic Object-Oriented Fault Injection Tool

GOOFI (Generic Object-Oriented Fault Injection Tool) is a fault injection tool that aims to have a user-friendly graphical interface and an architecture capable of accommodating new target systems and new fault injection techniques [27]. Taking advantage of the objected-oriented features of the Java language, GOOFI can easily accommodate new fault injection algorithms or new target systems. Since GOOFI was developed in Java language and all the data used by them is stored in a portable SQL-database, it makes the tool maintainable and portable between different host platforms, which was also an objective of the tool.

As already mentioned, GOOFI can easily accommodate new fault injection algorithms. The algorithms are built in an abstract class and each algorithm is composed of a set of calls to abstract methods. These abstract methods are implemented in the target system's classes, can be reusable by different fault injection techniques, and represents the steps of the fault injection. With that in mind, to add a new fault injection algorithm, the programmer must create a new method that uses the abstract methods as building blocks in the abstract class.

On another hand, if a programmer needs to use GOOFI with a target system that is not supported by the current version of the tool, a new class that extends the abstract one mentioned before must be created and must implement the abstract methods that will represent the steps of the fault injection to that specific target system. This makes easy the task of adding new fault injection algorithms and accommodating new target systems in the tool, allowing the use of GOOFI in various scenarios.

Focused on the fault injection techniques accommodated by GOOFI, the first version of the tool implements pre-runtime Software Implemented Fault Injection and Scan-Chain Implemented Fault Injection [27]. On the first one, faults are injected into the target system before it starts to execute and could be used to improve fault injection efficiency. In contrast, on the second one, faults are injected via boundary scan-chains and internal scan-chains that can be found on the modern microcontrollers, allowing to affect internal components of the integrated circuit and observe and monitoring the behavior of these target components. In short, GOOFI can inject single or multiple bit-flip faults.

2.3 Software Fault Tolerance Techniques

The use of fault injection tools is very useful to detect patterns of failures in a system. However, after they have been discovered, the probability of failure must be reduced in order to get some degree of dependability. Fault prevention techniques, such as code reviews, model checking, the use of strong-typed programming languages, and comprehensive software testing, among others, can be applied during the development phases. But these techniques cannot remove all points of failure in a system, and it is also impossible to control all variables of the system operation environment due to the high complexity of real software. With that in mind, fault tolerance techniques have the very important goal of providing more reliable software using as a basic assumption the idea that the software (and the system) is not perfect and may fail because of several reasons. In this context, fault tolerance techniques should detect errors (the consequences of faults) and provide means to tolerate such errors in order to avoid failures. It is worth mentioning that a failure is a visible manifestation of the component or system malfunction [12].

Fault prevention techniques can be applied in perfection. However, they cannot control the effect that space radiation has on the CubeSats. With that in mind, the use of fault tolerance techniques is inevitable to improve the reliability of such systems. Hardware fault tolerance techniques are very difficult to accommodate in CubeSats due to the limitations imposed by the CDS standard [1] (i.e., power consumption, size, weight, etc.). Hence, Software Implemented Fault Tolerance (SWIFT) can be used to minimize or even tolerate the induced faults caused by space radiation.

Fault tolerance is defined as the ability of a system to avoid service failures in the presence of faults [12]. The techniques that support fault tolerance can be divided into redundancy and error detection, and system recovery. A redundant system can be built with redundant hardware components that work simultaneously and a mechanism compares the results produced by the two components and declares an error if they differ. In addition, the redundancy can be also achieved by software, for example executing some functionality twice and voting the results to detect discrepancies (errors), or even to execute the functionality three times to mask the errors (i.e., tolerate hardware transient faults caused by radiation).

Error detection and system recovery techniques can be another alternative to handle transient faults, since after the detection of errors due to transient faults the recovery mechanisms should assure that the system is brought back to a consistent state, so it can continue delivering its service in a confident way. Error detection can be achieved by duplication and comparison – in practice with components or software redundancy. After that, the system recovery can be achieved using backward [43]–[45] or forward [43]–[45] recovery techniques. In both cases, the goal is to transform the erroneous state (at the error detection moment) into a new (correct) state from which the system can operate (in forward recovery), or a full recovery of the system to a state before the error occurrence (in backward recovery).

As already mentioned, CubeSats are very limited in terms of power consumption and size, which means that fault tolerance techniques that required more hardware components do not fit so well to increase the dependability of such systems. However, SWIFT can be very useful to mitigate or even tolerate the negative impact of SEU-induced faults in CubeSats boards. Obviously, these techniques are not “silver bullet”. With that in mind, one of the objectives of the work developed in the context of this thesis is to demonstrate that these techniques can achieve acceptable levels of dependability in CubeSats boards made with COTS with lower cost and effort.

Further, some software fault tolerance techniques are briefly explained in more detail to give some context to the reader.

Consistency Check

Consistency check uses a priori knowledge about the characteristics of the hardware components, the programming language used to develop the software, limited time to execute some operations, and other features of the system (hardware and software) and the application to verify its correctness. These verifications are usually implemented through assertions, which are software instructions that are placed in strategical places of the code to verify if any consistency property has been violated, leading to an error detection [43].

This technique can be applied to verify specific points such as data, address, or time consistency. Data consistency assertions aim to check the range of variables and the input parameters. Address consistency checks the addresses used by the processor to fetch instructions or access to data are within the correct addresses ranges for the different memory segments (i.e., code segment, data segment, stack segment), allowing for the detection of errors when the processor tries to accesses memory outside the declared memory segments. Time consistency aims to verify if some operation takes more time to run than it should, and when this happens, an error is detected.

Watchdog timers are a very popular example of time consistency techniques, which are normally used to recover the system after a crash. Nowadays, the watchdog timers are available built-in most microcontrollers [46]. In practice, the watchdog timer is a hardware timer circuit that must be periodically reset by software [47]. When the software does not suffer any deviation from its normal behavior, the timer will be reset before reaches zero. In contrast, if a hardware or software failure causes some deviation from the software normal behavior, the timer will not be reset and consequently, a processor interruption will be released. This interruption must be handled and the most common is to reset the system in order to prevent the delivery of wrong results and, also, hang situations.

Capability Check

Capability check is a fault tolerance technique implemented by software to assess that a component has the expected capability to execute its function. Often it is known as hardware testing[48].

The microprocessors have an internal component responsible to perform all arithmetic (e.g. Addition, Subtraction, Division, etc.) and logical operations (e.g. AND, OR, XOR, etc.). This unity is called Arithmetic Logic Unit (ALU) and can be tested in order to assure that is working well as expected. To do that, some capability check routines can be performed – for example, some specific instructions are executed with specific data and the result is compared with the expected one, and if the result differs an error in the ALU is detected.

Another example of this technique is the execution of some routines to test other components like memories. In this case, some data is written and read into some memory zone to ensure that the memories are performing well [49].

Software Diversity

Another fault tolerance technique is based on the execution of several software modules that are developed independently by different programming teams and the result of each one is voted. This technique is known as N-Version programming [50] and independent versions of design and code are developed from the same set of requirements. During the normal function of the system, the different versions are executed, and the result is voted by another component called voter. This voter must be capable to detect wrong outputs, preventing the propagation of erroneous values to the main output [51]. N-Version programming technique aims to prevent the occurrence of software errors that were been introduced during the development phase. The key principle of N-Version programming is to rely on team diversity (i.e., each version is developed by a different team in a totally intimate way) to avoid common mode faults. In fact, is not likely that different teams commit exactly the same software fault, thus the voting of the different versions will tend to detect and tolerate all software faults.

However, some researchers proved that the independence assumptions are not held by this technique [52], [53]. Knight & Leveson [52], [53] also discovered that different programming teams can make the same mistakes – since some problems are more difficult than others and humans tend to make the same mistakes in the same way. In addition to that, this technique represents more cost to the system under development.

A derivation of the previous technique is based on acceptance tests rather than a comparison of an equivalent version of the same functionality and is known as N-Self-Checking programming. In addition to the different versions of code, acceptance tests are performed on the output of each version of code. The voter is switched by a logic component that chooses the results from one of the programs that pass the acceptance tests.

Concerning the problem addressed in this thesis, related to the tolerance of transient hardware faults due to space radiation, the diversity features of N-Version programming are not particularly relevant, as diversity is totally aligned with the detection/tolerance of software faults and not transient hardware faults. Nevertheless, the execution of a given calculation more than one time followed by votation is a basic technique.

Error Detection

As already mentioned, error detection at software execution level is an important feature of fault tolerance techniques that are supported by the main techniques presented before and can be divided into two approaches: structural and behavior-based approaches [43].

The structural approach is mainly achieved by duplication and comparison. In practice, two or more copies of a software component that may be corrupted are executed and a mechanism that compares the output produced by each one and declares an error if the outputs differ – N-version programming is an implementation of this approach targeting software faults. It is unlikely that two or more copies will be corrupted together in the same way, making this approach efficient to detect for example SEU induced faults that corrupt data.

On the other hand, behavior-based approaches [43] are based on the execution of some routines that checks on the behavior of the target system – consistency and capability check techniques are based on this approach. The error detection is done by separate mechanisms like routines of code added to system code or hybrid approaches like watchdog timers that also require some specific hardware.

Both detection approaches can be applied to the development of software that runs on CubeSats in order to tolerate the effect of SEU-induced faults. With that in mind, the work developed under this thesis aims to prove that software implementing fault tolerance techniques should be used to develop CubeSat’s software.

After the detection of an error is mandatory to recover from that erroneous state to another one that can deliver a confident service. Whit that in mind, some recovery concepts and techniques are explored follow.

Error Recovery

Error recovery aims to transform a system state that contains one or more errors and faults into a state without faults that can deliver its service as well as would be expected [12]. It is possible to divide the error recovery into two groups of techniques. First, forward recovery [43]–[45] aims to transform the erroneous state into a new one from which the system can operate and deliver the service with confidence. Second, backward recovery [43]–[45] brings the system back to a state before the error occurrence. Techniques from both groups can be combined if the error persists after the application of one of them.

Forward error recovery requires the assessment of damages caused by the detected error or error propagation before the detection. An example of this application could be on a payload board of a CubeSat that occasionally missed the decoding of a message received from a sensor. The system can recover by skipping that decoding and moving on to the next message received. However, the risk of ignoring the error should be always assessed in order to control the error propagation and the well function of the entire system.

Within backward recovery, checkpointing and recovery blocks are two well-known techniques. The first one consists of making copies of the system's current state for possible use in a rollback in the future. These copies should be in stable storage in order to be available when the system is under the presence of a fault and should periodically be replaced with recent ones. Due to the few resources available in a CubeSat, this technique does not fit so well to recover from a fault. The overhead of saving the system state and the computation time between the checkpointing and the rollback can compromise the entire system. Despite this, checkpoint and backward recovery are highly successful in databases and transactional systems in general.

Recovery blocks were described in [54] and a system that uses this fault tolerance technique is made of blocks that represent various implementations of an algorithm. Each block contains at least a primary implementation, an alternative to that implementation, and a component that determines the correctness of the various implementations of each block which are called acceptance tests. The primary implementation is the block that is expected to execute without errors. The acceptance tests are performed at the end of the execution of each alternative in order to evaluate if that code was executed well. If the acceptance test fails, the state of the system before the first implementation started to be executed is restored and after, an alternative implementation is executed. In contrast, if the acceptance test pass, any alternative implementation available is discarded and the system continues the execution after the recovery block. Finally, these steps are made until an alternative implementation passes the acceptance tests. If all the alternatives fail the tests, the entire block has a failure, the algorithm could not be executed, and an error is signaled.

2.4 Concluding Remarks

This chapter starts by giving the reader basic background on small satellites and CubeSats. They represent an emergent trend and the number of launches of this space equipment has increased in the last years. With that in mind, making efforts to build more reliable satellites with the best cost possible is a major concern for the work developed under this thesis.

After that, the chapter surveys the use of fault injection in space applications. Fault injection is an attractive way of validating the resilience of space software applications and solutions. Performing fault injection during the development of such applications has an important role in the effectiveness validation that software fault tolerance techniques have when applied to the software that runs on top of such systems. With that in mind, the CubeSatFI fault injection platform aims to allow CubeSat developers to define fault injection campaigns that emulate the effects of space radiation, providing developers' teams with an effective tool to verify and validate CubeSat software in terms of SEU-induced faults

Since space systems made with COTS components can be affected by space radiation [9], it is very important to make the software that runs on top of them, more reliable. Thereby, software fault tolerance techniques seem the most promising approach to increase CubeSats resilience in terms of space radiation, while keeping the affordable budget, low energy, low mass, and easy to purchase hardware components of CubeSats. In this chapter, some of these techniques are briefly presented. The use of watchdog timers [14] can help to detect system crashes in order to put the system available again. Furthermore, the use of consistency routines to verify if the correctness output of specific software modules can help to detect errors concurrently, during the execution of the CubeSat software. Finally, replication and execution should be considered as a baseline SWIFI technique as it can be implemented using only software. Of course, replicating the entire system probably would have a huge impact on energy consumption. Instead, replicating just critical modules of software and comparing the result of them seems a good approach to detect some faults induced by space radiation.

Chapter 3
CubeSatFI Requirements and Architecture

One of the main objectives of this thesis is concerned with the development of a fault injector capable of reproducing the effects of hardware transient faults caused by space radiation in CubeSats boards. With that in mind, in collaboration with INPE, the CubeSatFI was designed to emulate Single Event Upsets (SEU) caused by space radiation in an exhaustive and fully automated way. In fact, the CubeSatFI intends to be compatible with any CubeSat board, once it provides a Joint Test Action Group (JTAG) interface.

As already mentioned before, this thesis is not only concerned with the development of a fault injector. In addition, the tool aims to be used in the evaluation of the reliability of CubeSats software against radiation, allowing developers to detect and cope with the negative impacts caused by space radiation with Software Implemented Fault Tolerance (SWIFT), increasing the reliability of CubeSats. Finally, to prove the effectiveness of such techniques, the CubeSatFI can be used again through the injection of a new campaign of faults.

This chapter aims to present the requirements and the architecture of the CubeSatFI fault injector.

3.1 Project Restrictions

INPE placed some technological restrictions that helped in the requirements elicitation. CubeSatFI must inject the faults using the IEEE 1149.1 standard for boundary-scan [38] available in CubeSat boards (the target systems of CubeSatFI). In order to abstract the communication between CubeSatFI and the Joint Test Action Group (JTAG) adapter [55], the tool must use the open-source library: Open On-Chip Debugger (OpenOCD) [56].

In addition, after the injected faults, the tool should save data to further analysis of the fault impact on the target system. During the development and testing of the tool, a payload board – Environment Data Collector (EDC) – from the CONASAT project was used as example of the target system. The EDC receives radio messages from ground stations, decodes those messages, and forwards them to the On-Board Computer (OBC) of the satellite. To simulate this communication between EDC and the OBC, an extra software component that emulates the OBC was developed, and the messages received are saved for further analysis.

3.2 Functional Requirements

The functional requirements aim to describe how CubeSatFI should work. With that in mind, CubeSatFI functionalities are organized in two groups:

1. Fault injection campaign generation and
2. Fault injection campaign execution.

The fault injection campaign generation allows the user to define controlled experiments through the specification of the number of faults to inject, type of faults to be injected, fault trigger conditions, among other things. The data describing each fault injection campaign is stored in a file and the user should be capable of importing the information from these files to define new fault injection campaigns.

The fault injection campaign execution controls automatically the fault injection process (i.e., no user intervention is needed) and executes all the steps required to inject each fault and collect the relevant data, according to the fault injection workflow. During the campaign execution, the OpenOCD server is launched on the Host PC and it is responsible for receiving all the instructions from the CubeSatFI, forwarding them to the target system through JTAG, and, consequently, receiving the respective responses.

The target system behavior after the injection of each fault is collected and saved in a file. The information collected depends on the actual scenario in which the target system is being used and the specific purpose of the fault injection campaign. Since the results obtained by fault injection depend on the software running on top of COTS, the collection of results should be defined taking into account the specific functionalities of the system under testing and the testing objectives. The file with the fault injection results stored at the end of each campaign could be analyzed using external statistical tools such as Excel or R. This analysis is a functionality not supported by CubeSatFI.

Faults are described by two groups of parameters, following well-established practice in the fault injection area:

- Fault type: indicates the exact location of the fault in the target system and the number of bits affected (single bit-flip or multiple bit-flips). Only bit-flip faults are considered since this is a well-established fault model for hardware faults induced by SEU [5], [6].
- Fault trigger: indicates the exact moment/conditions when the fault should be injected.

On CubeSatFI platform, faults should be injected in any bit of any register of the target processor, emulating SEU, and radiations bursts through single bit-flip or multiple bit-flips, respectively. Moreover, triggers should consider two principal domains. The first is the time domain where the injection of faults occurs randomly in the injection widows presented in Figure 3. This is the basic method to get the statistical effects of faults induced by space radiation, as the injected faults are distributed at random in both the space (i.e., registers and register bits) and time domains. Further, the second domain is location-based, since faults should be injected when the program executes a given instruction according to the workflow presented in Figure 4. This second trigger allows to design more specific and detailed campaigns to analyze the behavior of the target system in specific moments (i.e., when specific instructions are executed).

Figure 3 presents the principal steps of the injection workflow for time-based injection. Resetting the target is the first step of each injection run (an "injection run" can be defined as a sequence of steps needed to inject a fault and collect results on the impact of such fault) to assure that the results in each fault are not "contaminated" by the effects [27], [32] of the previous fault. Once reset the target system, and until the end of each injection run, there are some important moments depicted in Figure 3 that require some explanations:

- **Start of the Injection Window (T0)**: From this moment the fault can be injected. This time indicates the start of a run injection window.
- **Injection (Tinj)**: When the fault is actually injected into the target system. In time-based fault triggers, this moment is calculated randomly and corresponds to a time between the T0 and T1.
- **End of the Injection Window (T1)**: The fault can be injected until this moment. This time indicates the end of the run injection window.
- **End of the Injection Run (Tend)**: Indicates the end of the injection run. The time between T1 and Tend (Tend -T1) is exclusive to save information about the target system's behavior after it has worked for some time with the fault.

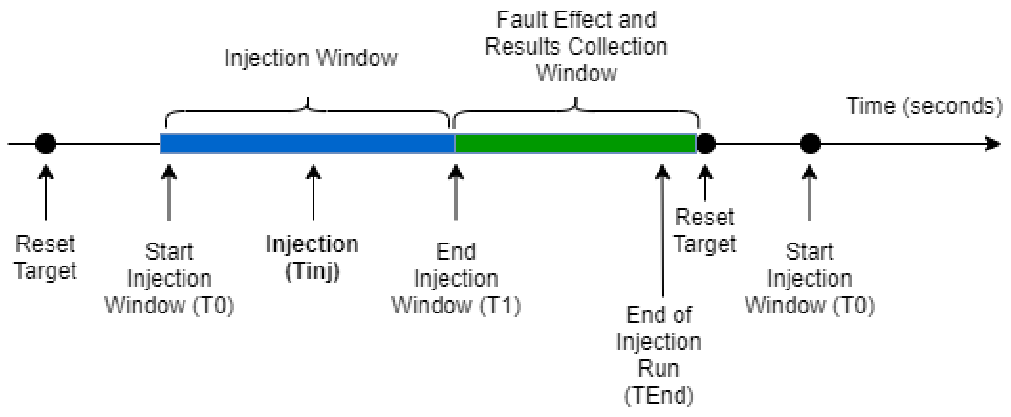

Injection Window
Fault Effect and Results Collection Window
Time (seconds)
Reset Target
Start Injection Window (T0)
Injection (Tinj)
End Injection Window (T1)
End of Injection Run (TEnd)
Reset Target
Start Injection Window (T0)

Figure 3 - Time-based Fault Injection Workflow

The collection of the results is the last step of one injection run and is done after some time interval to assure that the system works for some time after the injection of the fault, to be possible to measure the impact of the fault on the target system. This information depends always on the target system and the software that runs on top of it. Hence, the software module responsible for the collection of the results impact should be developed for each target system in specific.

Figure 4 presents the presents the principal steps of the injection workflow for location-based injection. Similar to the time-based fault injection workflow, the first step consists of resetting the target system to assure that the results in each fault are not "contaminated" by the effects of the previous one. After that, a breakpoint is set in a given location of the code and the target is reset again to assure that the target system executes all the code until the breakpoint. Hence, when the breakpoint is triggered, the fault is injected, and the target system will run with the fault for a period of time. During this time, data is collected for further analysis of the impact of the fault.

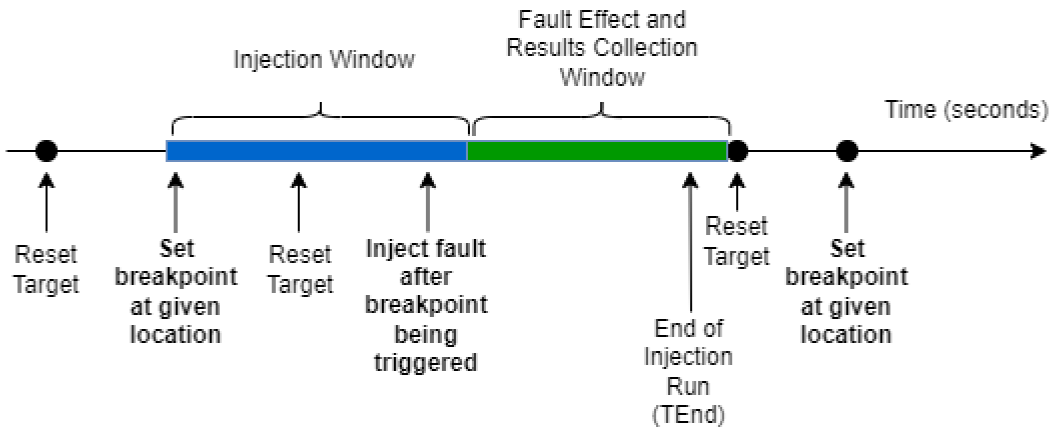

Injection Window
Fault Effect and Results Collection Window
Time (seconds)
Reset Target
Set breakpoint at given location
Reset Target
Inject fault after breakpoint being triggered
End of Injection Run (TEnd)
Reset Target
Set breakpoint at given location

Figure 4 - Location-based Fault Injection Workflow

To describe the high-level functionalities of CubeSatIF mentioned above, a use case diagram was created. Figure 5 contains the legend of the elements used in the diagram and Figure 6 shows the high-level use case diagram that summarizes the functional requirements of CubeSatFI.

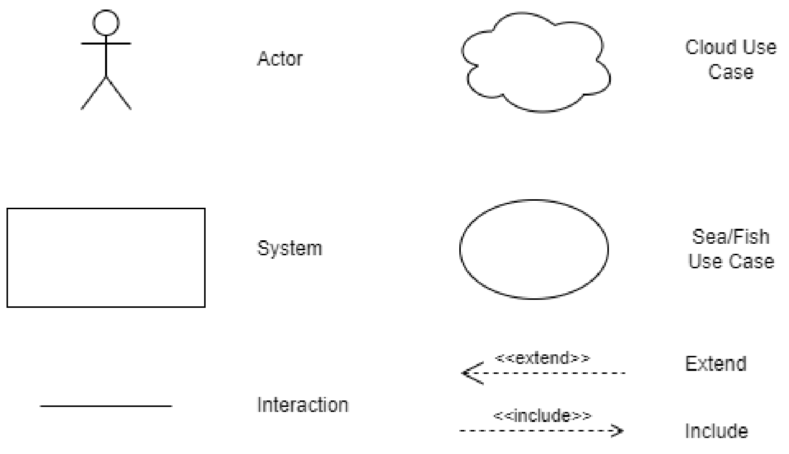


Figure 5 - Legend of the elements used in the use case diagram

In short, the “Actor” represents the user, or the users of the system described on the use case diagram. The “System” can be the system that is described by use cases or an external system that communicates with the described one. The “Interaction” represents a liaison between an actor and a use case. Hence, a “Cloud Use Case” is a summary of a functionality that can be divided into a set of “Sea/Fish Use Case” that represents a user goal. Finally, the different use cases can have dependencies between them. An “Include” dependency is the invocation of a use case by another one. In contrast, an “Extend” dependency is an alternate course of the base use case.

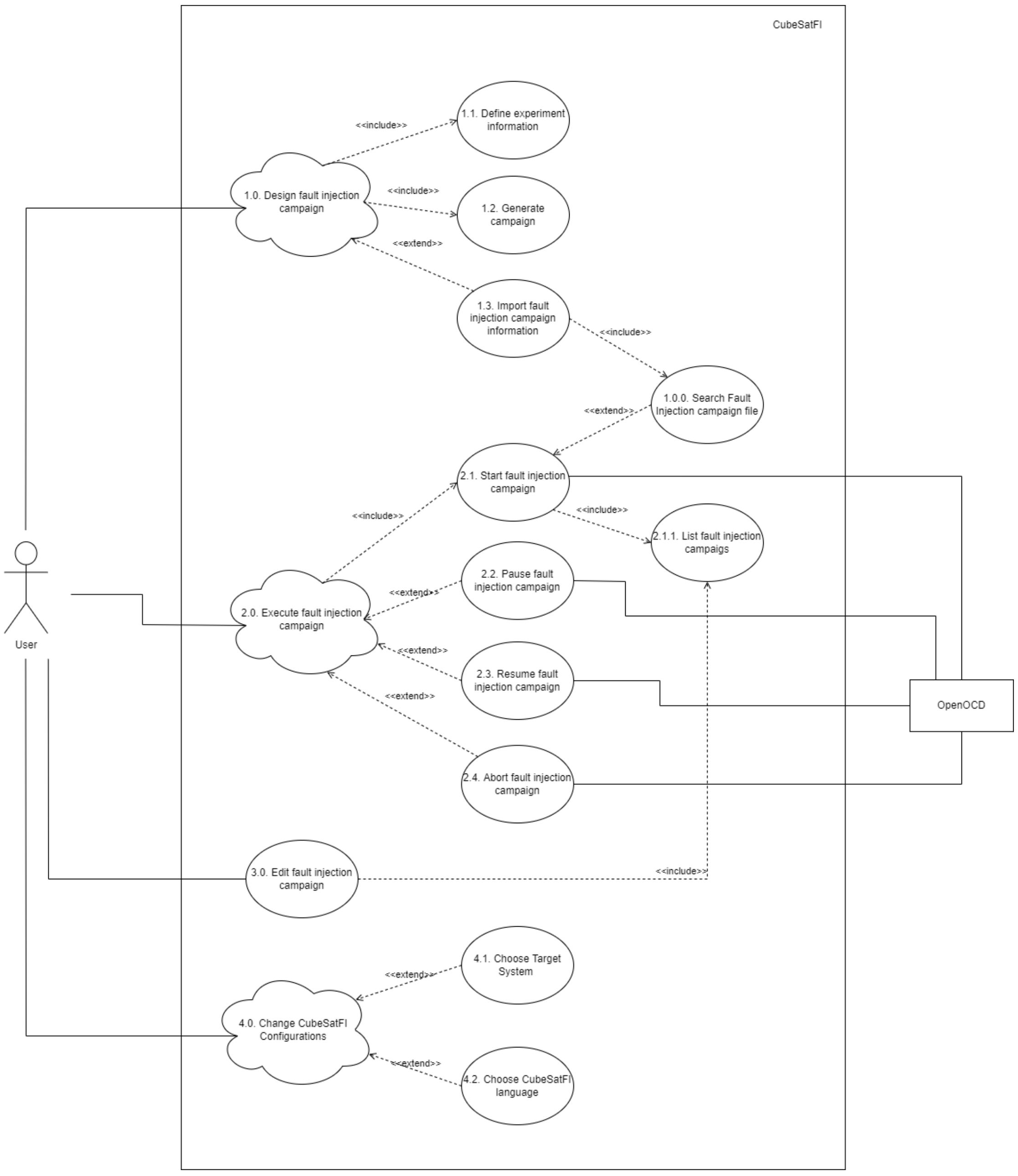

CubeSatFI
1.0. Design fault injection campaign
1.1. Define experiment information
1.2. Generate campaign
1.3. Import fault injection campaign information
1.0.0. Search Fault Injection campaign file
2.0. Execute fault injection campaign
2.1. Start fault injection campaign
2.1.1. List fault injection campaigs
2.2. Pause fault injection campaign
2.3. Resume fault injection campaign
2.4. Abort fault injection campaign
3.0. Edit fault injection campaign
4.0. Change CubeSatFI Configurations
4.1. Choose Target System
4.2. Choose CubeSatFI language
<<include>>
<<extend>>
User
OpenOCD

Figure 6 - Use Case Diagram of the High-Level Functional Requirements

The use cases represented in Figure 6 are described in detail in Appendix A – CubeSatFI Functional Requirements.

3.3 Non-Functional Requirements

Functional requirements define how the system should work. In contrast, non-functional requirements define constraints that affect how the system should perform. In fact, a system can work if non-functional requirements are not met, but maybe it cannot meet the stakeholder’s expectations and the business needs.

In order to meet the project needs, two main non-functional requirements arise – compatibility, and modifiability –, and are explained following.

Concerning compatibility, CubeSatFI intends to be compatible with a wide set of processors and not limited to performing fault injection just on one specific target processor, allowing the use of the tool in a wide range of CubeSats projects. To accomplish that CubeSatFI will use the IEEE 1149.1 standard for boundary-scan available in most modern processors used on CubeSats. If CubeSats boards have the JTAG port, they are compatible with CubeSatFI. For example, ARM Cortex-M3[6] and ARM Cortex-M4[7] not only have a JTAG port, but they also have the same registers map, which means that CubeSatFI can be used on both without any modification. These two processors are used in the Environment Data Collector (EDC) payload board and on the On-Board Computer (OBC) of the CONASAT CubeSat, respectively.

On another hand, modifiability should be contemplated on CubeSatFI to be easy to accommodate new target systems, fault types, fault triggers, and other features like other idioms. This can be achieved using an oriented-objected language to take advantage of some characteristics of these type of languages, such as inheritance and polymorphism. Abstract classes and interfaces should be used to define the steps of the fault injection and concrete classes should extend them and implement their defined methods. These implementations are different taking into account different targets, fault types, and triggers, which allows modifying the fault injector to a specific need.

Since the CubeSatFI is being developed in the context of international projects, already referred in the introduction of this document, and the main stakeholder of the application is INPE, there was a clear requirement for the tool to be multi-language, Portuguese and English. This requirement was handled in a more general way, as CubeSatFI can be easily extended to other languages, in addition to the Portuguese and English.

3.4 Platform Architecture

Figure 7 shows the CubeSatFI fault injector setup. The Host PC runs the injection tool that uses the OpenOCD [56] to communicate with the target system and perform the injection of the faults using the IEEE 1149.1 standard for boundary-scan [55] available in CubeSat boards. The CubeSatFI running on the host PC uses the debugging and boundary-scan features available through the JTAG adapter [55] to get access to the processor registers and other internal structures via the JTAG Test Access Port (TAP) of the target system.

[6]https://developer.arm.com/documentation/dui0552/a/the-cortex-m3-processor/programmers-model/core-registers (accessed Jan. 22, 2022)
[7] https://developer.arm.com/documentation/dui0553/b (accessed Jan. 22, 2022)

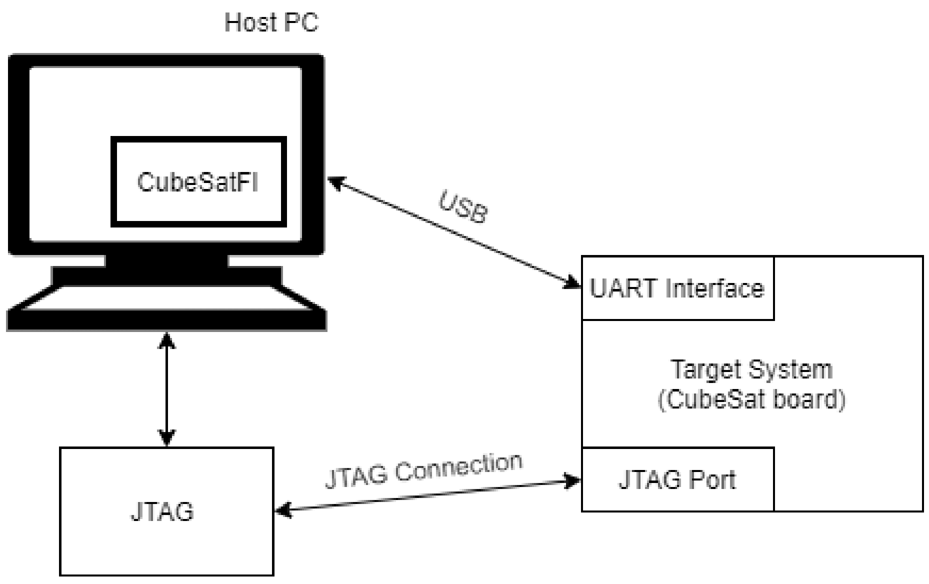


Figure 7 - CubeSatFI fault injection setup

The basic idea to perform fault injection using the IEEE 1149.1 standard for boundary-scan consists of using JTAG commands through OpenOCD to interrupt the normal execution of the program running on the target system and to get a copy of the internal state of the processor and other internal data included in the scan chain of the target system. Then, one or more bits of such internal state can be corrupted (according to the fault models described above) and the internal state is put back again, including the error caused by the emulated fault. After that, the normal execution of the software is resumed and the impact of the emulated fault in the target system is evaluated. The fault injection algorithm is discussed further on in more detail. This approach has already been used successfully to inject faults [27] and has the advantage of using the features available in the target system for testing purposes to inject faults with minimal perturbation of the system (other than the injected fault). Since most of the CubeSat boards include JTAG and TAP port, the proposed CubeSatFI can be used in most of the CubeSat satellites.

Hence, the Java programming language was chosen to develop the CubeSatFI fault injector in order to take advantage of the object-oriented properties allowing the easy modifiability of the tool (i.e., adding new fault types, new fault triggers, new target systems, new languages, among others). Taking into account the user interface, the JavaFX Framework[8] was used in the development of the graphical interface, once it allows the development of modern, clean, and user-friendly applications, including desktop applications.

Figure 8 presents a diagram that illustrates the CubeSatFI architecture. As already mentioned, the fault injector runs on a host computer that is also responsible to simulate any device that is part of the CubeSat ecosystem in order to get a test

[8] https://openjfx.io/ (accessed Jan. 18, 2022).

environment closer to the real one (e.g., the simulation of a ground station that sends/receives messages to and from the CubeSat). Besides, the tool intents to support the addition of new fault types and target systems troughs the use of object-oriented properties of the Java programming language. Jointly, the graphical user interface (GUI) must be adapted to accommodate new fault types or target systems. The GUI interacts with the two main components, briefly explained below:

- **Campaign Generation Component**: Controls all the generation process of the campaigns by receiving the inputs from the GUI, generating the faults, and saving them into a file.
- **Campaign Execution Component:** Controls all the fault injection process by reading the faults from a file, initializing the OpenOCD server, and controlling the injection of each fault.

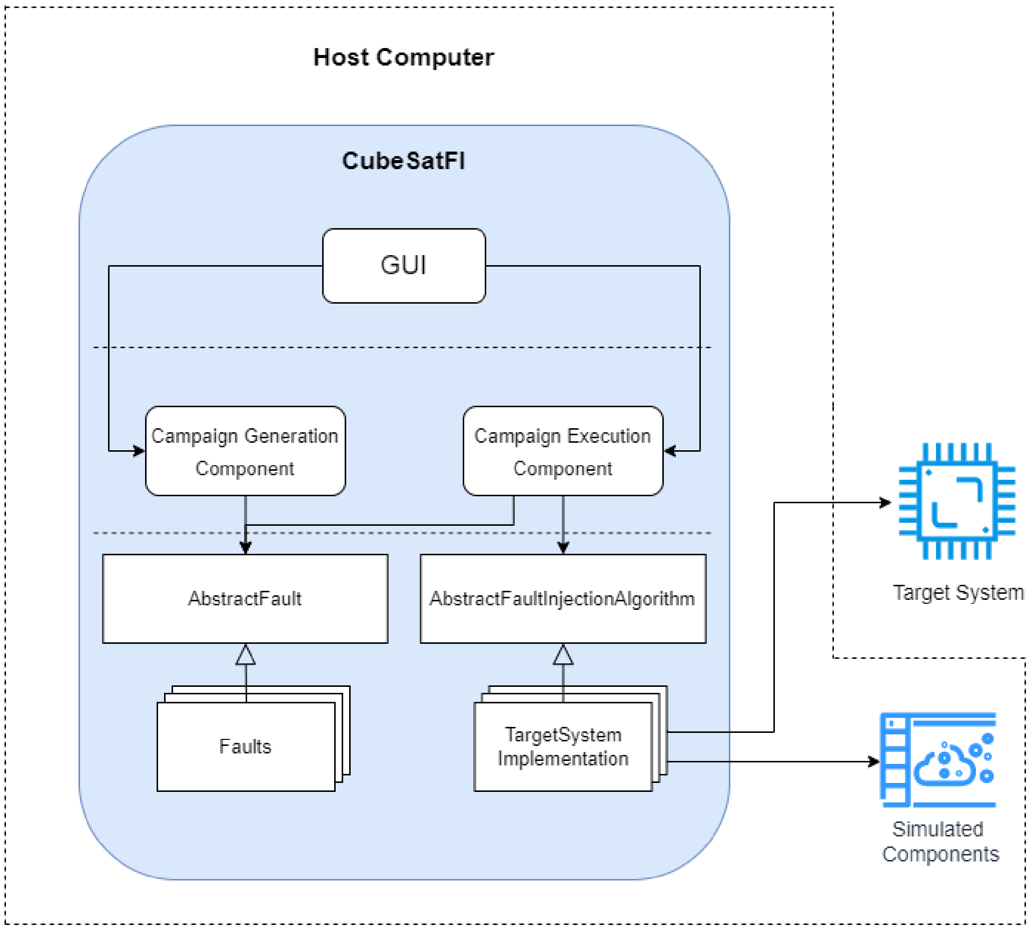
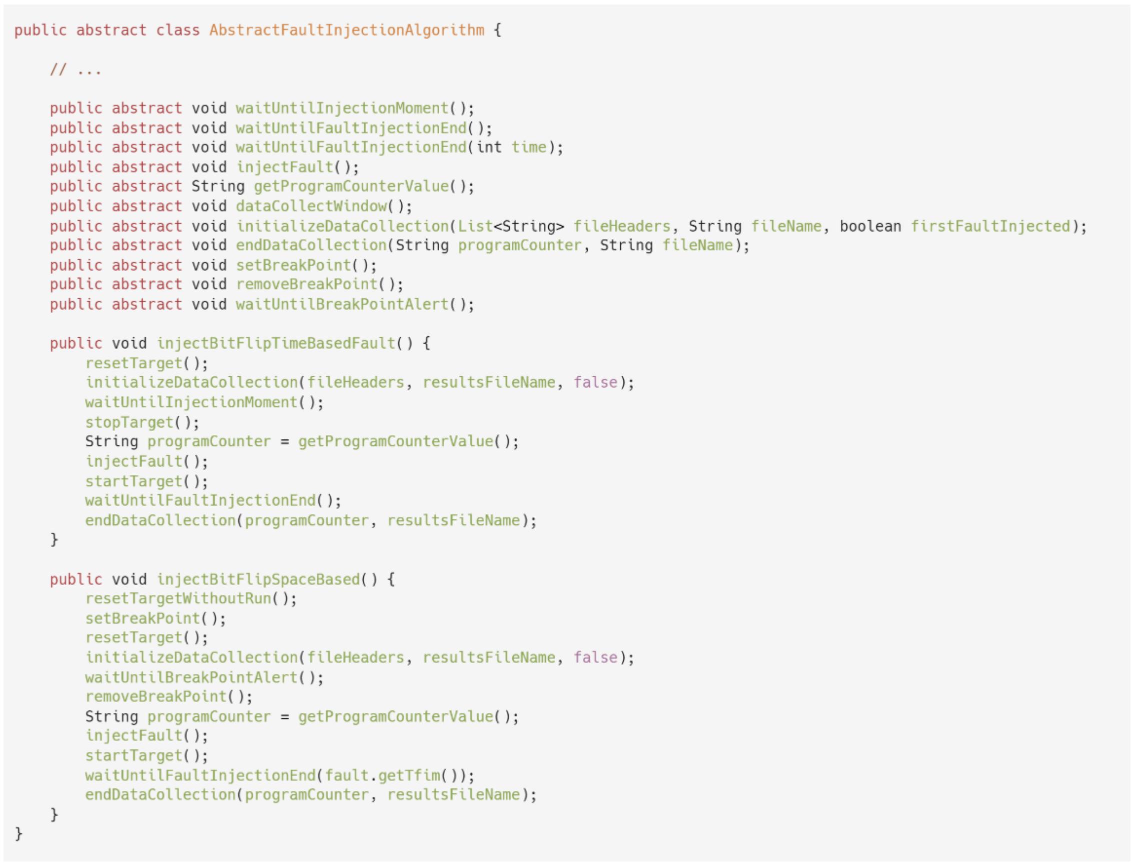

Host Computer
CubeSatFI
GUI
Campaign Generation Component
Campaign Execution Component
AbstractFault
AbstractFaultInjectionAlgorithm
Faults
TargetSystem Implementation
Target System
Simulated Components

Figure 8 - CubeSatFI Architecture

To add new fault types, the programmer must create a new Fault class that inherits the AbstractFault class (Figure 8) and implement the abstract methods that will be used on that fault type. In fact, the programmer just needs to implement the abstract methods used by that fault type. Likewise, the same situation is applied when the programmer wants to add a new target system to CubeSatFI. The programmer must create a TargetSystem class that inherits AbstactFaultInjectionAlgorithm and implement the abstract methods used by the fault injection algorithms on that specific target (If the target system is different but have the same registers map, is not necessary to implement a new target system). The abstract methods work as building blocks which means that the fault injection algorithms are built through the combination of different abstract methods. Figure 9 shows two fault injection algorithms available and defined in the AbstractFaultInjectionAlgorithm class. Both are constructed with abstract methods that are further implemented in the concrete TargetSystem classes. Each target system implements a version of each step of the fault injection algorithm.



```java
public abstract class AbstractFaultInjectionAlgorithm {

    // ...

    public abstract void waitUntilInjectionMoment();
    public abstract void waitUntilFaultInjectionEnd();
    public abstract void waitUntilFaultInjectionEnd(int time);
    public abstract void injectFault();
    public abstract String getProgramCounterValue();
    public abstract void dataCollectWindow();
    public abstract void initializeDataCollection(List<String> fileHeaders, String fileName, boolean firstFaultInjected);
    public abstract void endDataCollection(String programCounter, String fileName);
    public abstract void setBreakPoint();
    public abstract void removeBreakPoint();
    public abstract void waitUntilBreakPointAlert();

    public void injectBitFlipTimeBasedFault() {
        resetTarget();
        initializeDataCollection(fileHeaders, resultsFileName, false);
        waitUntilInjectionMoment();
        stopTarget();
        String programCounter = getProgramCounterValue();
        injectFault();
        startTarget();
        waitUntilFaultInjectionEnd();
        endDataCollection(programCounter, resultsFileName);
    }

    public void injectBitFlipSpaceBased() {
        resetTargetWithoutRun();
        setBreakPoint();
        resetTarget();
        initializeDataCollection(fileHeaders, resultsFileName, false);
        waitUntilBreakPointAlert();
        removeBreakPoint();
        String programCounter = getProgramCounterValue();
        injectFault();
        startTarget();
        waitUntilFaultInjectionEnd(fault.getTfim());
        endDataCollection(programCounter, resultsFileName);
    }
}
```

Figure 9 - AbstactFaultInjectionAlgorithm class

3.5 Concluding Remarks

This chapter presents to the reader the project restrictions, the requirements, and the architecture of the CubeSatFI. CubeSatFI intends to be a fault injector compatible with all types of CubeSat boards, provided they have JTAG Boundary Scan. With that in mind, the communication with the JTAG interface is supported by the JTAG adapter and the OpenOCD library.

In addition, the tool was designed to support the addition of new fault types and target systems. Therefore, the programmer should inherit an abstract class and implement concrete methods that compose the new fault type or the steps for the fault injection algorithms on a specific target system respectively. Taking advantage of the object-oriented properties of the Java language allows to modify and adapt the tool in order to accommodate new fault types and target systems in an easy and efficient way.

The next chapter presents the fault injector and all its functionalities.

Chapter 4
CubeSatFI Functional View

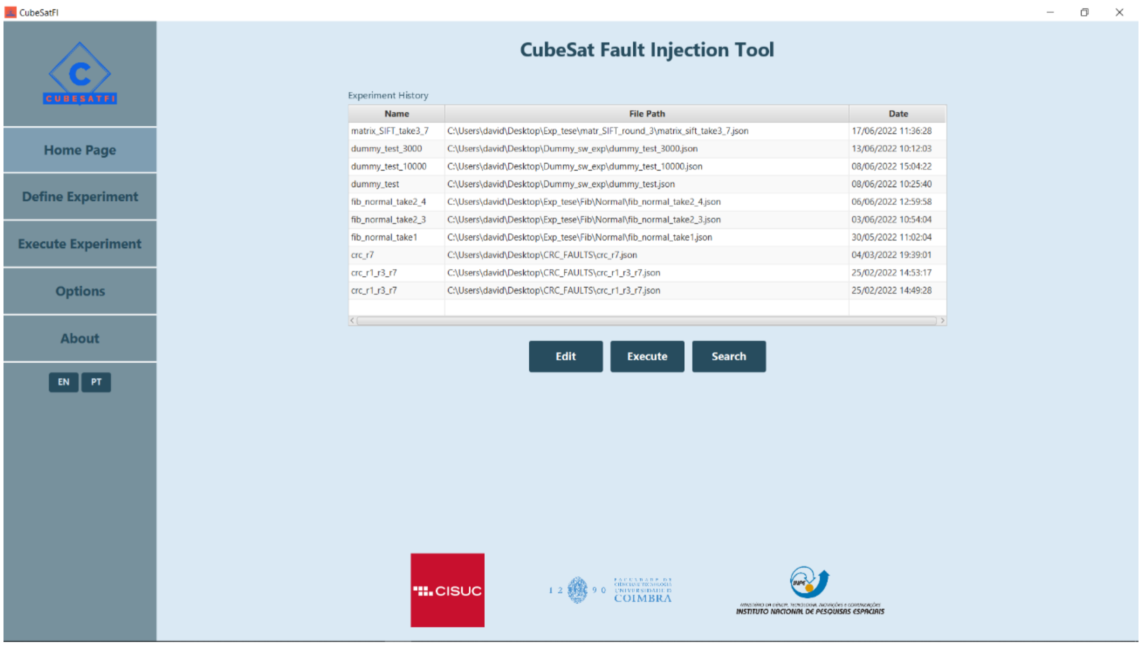
CubeSatFI aims to be used for verification and validation of software that runs on top of CubeSat boards allowing the easy definition of fault injection campaigns that emulate the effects of space radiation.

This chapter presents the CubeSatFI main screens and functionalities. In addition, presents a preliminary experiment and a correspondent analysis of the results obtained on a fault injection campaign performed on a payload board from the CONASAT project [8].

4.1 The Fault Injector

Figure 10 shows the home page screen that contains a table with the history of the last generated fault campaigns. The user can select one of them and execute the fault injection campaign by clicking on the central button “Execute” or the user can click on the left-hand button “Edit” if they intend to edit some parameter of the campaign. Furthermore, the user can add a fault campaign record to the history table by clicking on the button “Search”. A window will be open, and the user just needs to find and select the intended fault campaign configuration file.

CubeSatFI intends to be a clean and intuitive tool concerning its usability. With that in mind, a vertical menu is always visible as shown in the screen presented in Figure 10 and the selected page is highlighted in a lighter color. The user can also change the language in which the tool is presented by clicking on the buttons at the bottom of the menu (“EN” to choose English or “PT” to choose Portuguese). On the other pages, the user cannot change the language of the tool. This last functionality is only available on the home page.

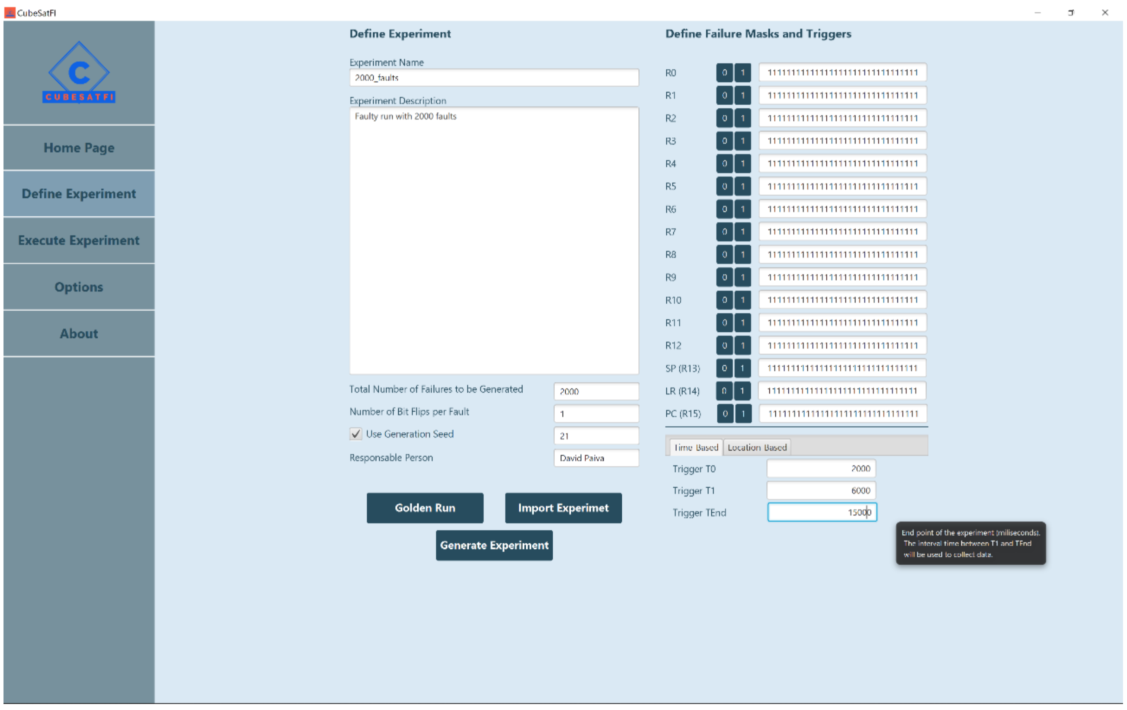


Experiment History

Name	File Path	Date
matrix_SIFT_take3_7	C:\Users\david\Desktop\Exp_tese\matr_SIFT_round_3\matrix_sift_take3_7.json	17/06/2022 11:36:28
dummy_test_3000	C:\Users\david\Desktop\Dummy_sw_exp\dummy_test_3000.json	13/06/2022 10:12:03
dummy_test_10000	C:\Users\david\Desktop\Dummy_sw_exp\dummy_test_10000.json	08/06/2022 15:04:22
dummy_test	C:\Users\david\Desktop\Dummy_sw_exp\dummy_test.json	08/06/2022 10:25:40
fib_normal_take2_4	C:\Users\david\Desktop\Exp_tese\Fib\Normal\fib_normal_take2_4.json	06/06/2022 12:59:58
fib_normal_take2_3	C:\Users\david\Desktop\Exp_tese\Fib\Normal\fib_normal_take2_3.json	03/06/2022 10:54:04
fib_normal_take1	C:\Users\david\Desktop\Exp_tese\Fib\Normal\fib_normal_take1.json	30/05/2022 11:02:04
crc_r7	C:\Users\david\Desktop\CRC_FAULTS\crc_r7.json	04/03/2022 19:39:01
crc_r1_r3_r7	C:\Users\david\Desktop\CRC_FAULTS\crc_r1_r3_r7.json	25/02/2022 14:53:17
crc_r1_r3_r7	C:\Users\david\Desktop\CRC_FAULTS\crc_r1_r3_r7.json	25/02/2022 14:49:28

Figure 10 - Home Page Screen

Figure 11 shows the campaign definition screen where the fault injection campaigns are defined. In the current version of CubeSatFI, the user can define fault injection campaigns that emulate SEU affecting processor registers of the processor of the target CubeSat board. On the central area of the form shown in Figure 11, the user defines the experiment name, description, number of faults to be injected in the campaign, number of bitflips per fault, the seed to use in the random number generation (this is important to decide whether the experiment is reproducible or not), and the name of the person responsible for the fault injection campaign. On the right-hand side of the form, the user defines the masks to apply to the target processor registers that define the register and/or registers that can be selected to inject faults and the bits inside each register that can be affected by the faults. At the bottom on the right-hand side of the form, the user selects the fault triggers (in time or location domain) and defines the timing for the different moments of the fault injection workflow, as defined in Figure 3, or defines the location where the fault is injected. After providing all this data, the user can click on the button “Generate Experiment” to ask CubeSatFI to generate the description of the faults, which are stored in a file with all the information about the injection campaign.



Define Experiment

Experiment Name: 2000_faults

Experiment Description: Faulty run with 2000 faults

Total Number of Failures to be Generated	2000
Number of Bit Flips per Fault	1
✓ Use Generation Seed	21
Responsable Person	David Paiva

Define Failure Masks and Triggers

R0	11111111111111111111111111111111
R1	11111111111111111111111111111111
R2	11111111111111111111111111111111
R3	11111111111111111111111111111111
R4	11111111111111111111111111111111
R5	11111111111111111111111111111111
R6	11111111111111111111111111111111
R7	11111111111111111111111111111111
R8	11111111111111111111111111111111
R9	11111111111111111111111111111111
R10	11111111111111111111111111111111
R11	11111111111111111111111111111111
R12	11111111111111111111111111111111
SP (R13)	11111111111111111111111111111111
LR (R14)	11111111111111111111111111111111
PC (R15)	11111111111111111111111111111111

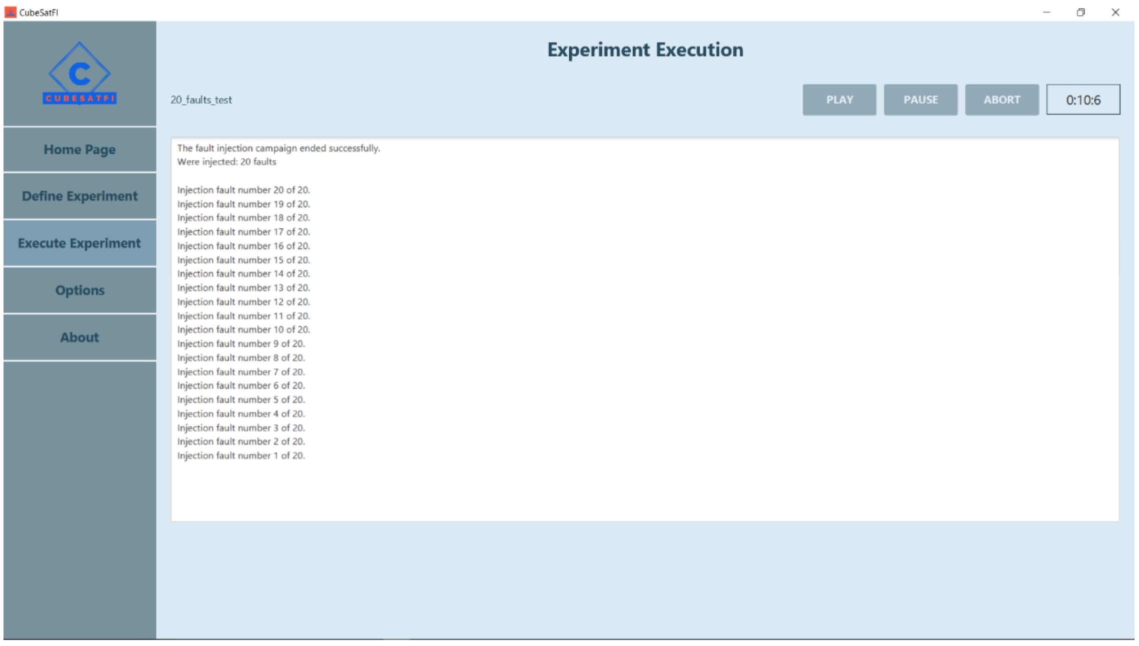
Trigger T0	2000
Trigger T1	6000
Trigger TEnd	15000

End point of the experiment (milliseconds). The interval time between T1 and TEnd will be used to collect data.

Figure 11 - Campaign Definition Screen

The button “Import Experiment” allows the user to select a file describing a fault injection campaign previously defined and loads such file. After loading the file (or after defining a new fault injection campaign), the user can always change any of the parameters that define the fault injection campaign, or can select the execution of a “Golden Run” at the bottom of the central part of the screen. The golden run simply runs the workload one or more times to record the correct behavior of the target system. In practice, the golden run is similar to a fault injection run, with the capital difference that no fault is injected. The behavior of the target system recorded during the execution of the golden run will be used later on to evaluate the failure modes of the target system after each injection run.

After defining a fault injection campaign, the user can select “Execute Experiment” on the left-hand side menu in order to start a fault injection campaign. As already mentioned, this step is fully automatic, follows the workflow presented in Figure 3, and is totally controlled by the CubeSatFI. The central window in Figure 12 displays contextual information about the fault injection process, namely the number of the fault currently being injected. During the execution of the fault injection campaign, the user can simply pause the fault injection process by clicking on the button “Pause” (the system will suspend the fault injection process after completing the fault currently being injected). To resume the injection process, later on, the user should click on the "Start" button. The user can also abort the fault injection campaign by clicking on the button “Abort”.



20_faults_test

0:10:6

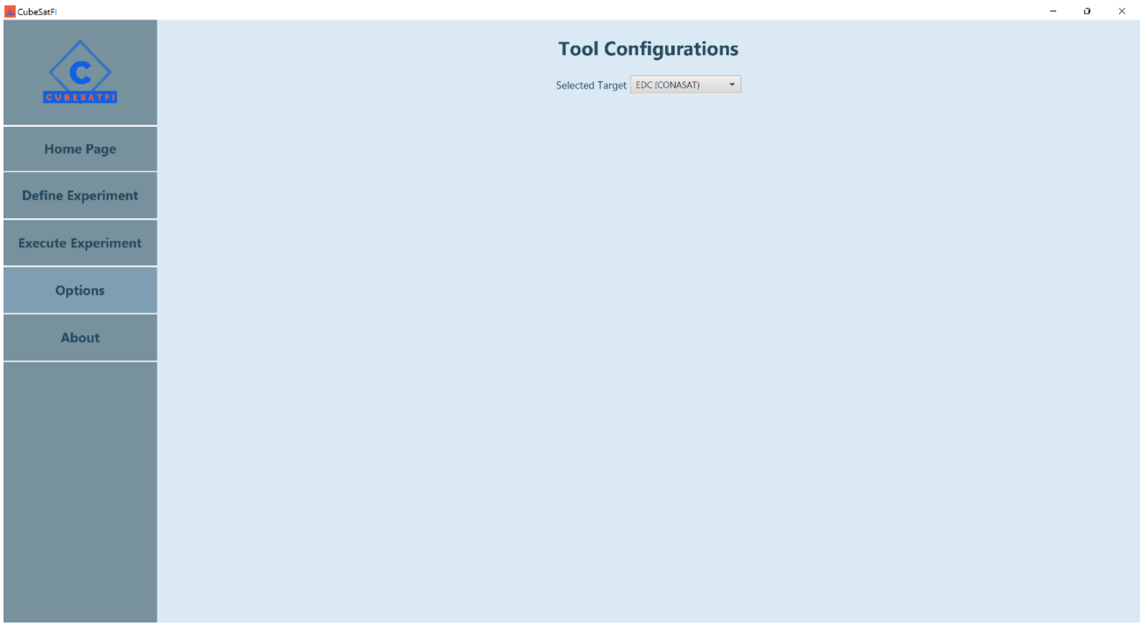
```
The fault injection campaign ended successfully.
Were injected: 20 faults

Injection fault number 20 of 20.
Injection fault number 19 of 20.
Injection fault number 18 of 20.
Injection fault number 17 of 20.
Injection fault number 16 of 20.
Injection fault number 15 of 20.
Injection fault number 14 of 20.
Injection fault number 13 of 20.
Injection fault number 12 of 20.
Injection fault number 11 of 20.
Injection fault number 10 of 20.
Injection fault number 9 of 20.
Injection fault number 8 of 20.
Injection fault number 7 of 20.
Injection fault number 6 of 20.
Injection fault number 5 of 20.
Injection fault number 4 of 20.
Injection fault number 3 of 20.
Injection fault number 2 of 20.
Injection fault number 1 of 20.
```

Figure 12 - Campaign Execution Screen

Concerning the usability of the tool, a menu with the main features of the CubeSatFI is always present on the left-hand side of the screen as already mentioned. The feature currently selected is highlighted in a lighter color, making it easy to identify the selected functionality. Furthermore, to improve the user's experience, tooltips are available in most fields and buttons. These messages are displayed whenever the user puts the mouse over them and an example is shown in Figure 11 in the bottom right-hand corner of the form.

Figure 13 shows the options screen where the user can adjust the settings of the CubeSatFI. The current version of the tool only made available the option of selecting the current target system. Once selected the target system, the CubeSatFI will always assume that target on the generation and execution of fault injection campaigns. In the future, other configurations can be added to the tool, but for now, the selection of the target system is the only one that is necessary.



Selected Target: EDC (CONASAT)

Figure 13 - Options Screen

4.2 Preliminary Experiment

In an effort to present CubeSatFI to the scientific community an article [10] that presents the CubeSatFI, and a first real example of its usage was submitted and accepted to the 2021 Latin-American Symposium on Dependable Computing (LADC) and published in the conference proceedings by the Institute of Electrical and Electronics Engineers (IEEE) at IEEE Xplore. The paper can also be found in Appendix B - Fault injection platform for affordable verification and validation of CubeSats software.

In this preliminary experiment, the Environmental Data Collector (EDC)[9] was selected as the target system for this experiment. The EDC is a CubeSat payload for the Brazilian Environmental Data Collection System that is going to be used in all the nanosatellites from the CONASAT project [8].

Environmental Data Collector (EDC)

The EDC is a multi-user RF receiver for a satellite message forwarding system. These systems offer sensor data transmission services in remote areas, such as environmental monitoring, wildlife tracking, vessel tracking, among others, with the lowest structural cost.

These systems consist of ground platforms (GP), a satellite constellation, one or more receive stations (RS) and a data distribution center (DDC). The GPs transmit local sensor data through periodic messages, coded in RF burst transmissions. The satellite relays the received GP messages to a RS, when possible. Finally, the RSs transmits the GP messages to the DDC, which provides the cloud service for the system users. In this context, the

[9] http://www.inpe.br/nordeste/projetos/edc.php (accessed Nov. 25, 2021).

EDC is the satellite payload that receives and decodes the GP messages. It does not have a transmitter. Therefore, the satellite onboard computer (OBC) must read the received message from the EDC and forward them to an RS multiplexing the data in one of its telemetry channels. Figure 14 illustrates the Brazilian Environmental Data Collection System described above, to which the EDC belongs.

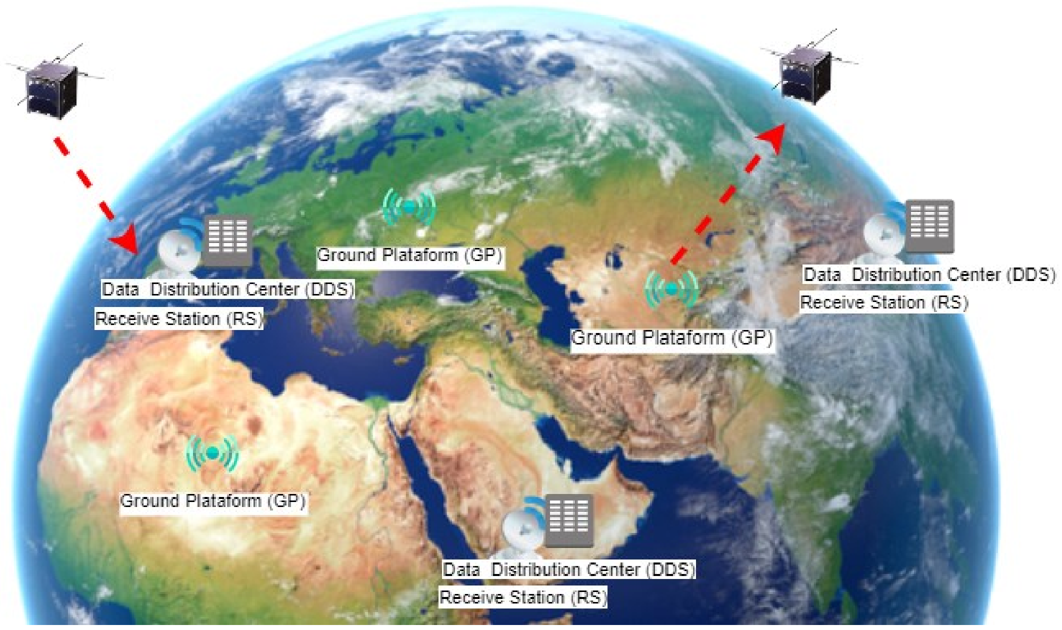


Figure 14 - Brazilian Environmental Data Collection System

The EDC has a RF-Front-End unit that digitalizes the received RF signal and a processing unit that configures the RF-Front-End at system startup, decodes the received GP signals, and provides the interface with the OBC through a serial port. Besides the decoded messages, the EDC also provides housekeeping information to the OBC, such as supply current and voltage sensor measures, elapsed time since the last system reset, and others. The processing unit is implemented in an SoC FPGA (System on Chip - Field Programmable Gate Array), which has an internal hardwire microcontroller based on a Cortex-M3 processor. The signal decoding processing is splitted between the FPGA (hardware) and the processor (software), while the OBC interface and RF-Front-End configuration are fully implemented in software. The EDC software runs on top of the multi-task based FreeRTOS real-time operating system[10].

When the EDC is turned on, the OBC initializes it by sending the Real-Time Clock. A housekeeping frame must be requested for checking the temperature, electrical current, and electrical voltage sensors. This frame also indicates if the RF-Front-End configuration was successful. After this verification, signal decoding must be enabled, which starts disabled by default.

The OBC must periodically request the status of the decoded message buffer, in order to request the reading of the GP packages, if there are any messages. These packages

[10] https://www.freertos.org/RTOS.html (accessed Nov. 30, 2021).

have a variable length and are composed of a tag RTC time the message was received, a code that indicates an error in the decoding process, the frequency and amplitude of the received signal, the message length, the variable-length GP message, and a checksum byte, at each reading of a GP package, the OBC must send a command to remove the package from the buffer, allowing the reading of another package. For telemetry, it is expected to send a housekeeping frame followed by a sequence of GP packages.

Experiment Setup

The goal of this first experiment is to demonstrate the use of CubeSatFI to evaluate the impact of SEU-induced faults in the EDC CubeSat board. With that in mind, a fault injection campaign with 2000 faults injected at random has been defined, since the space radiation tends to affect the board in a random way. Faults are injected into any register of the processors, selected at random, and within the selected register for a given fault, the bit of the register affected by the fault was also selected at random. All the faults are single bit-flip faults, as this model is widely accepted as a realistic emulation of SEU faults. The trigger of each fault is also defined at random within the injection window (see Figure 3). The injection window interval was defined as between 2 and 4 seconds.

After the end of the injection window, the target system will be running to evaluate the effects of the fault for a period of 6 seconds. The messages decoded by the EDC are saved in a file (obviously, in some cases the injected fault causes the EDC to crash, and the message decoding is interrupted).

Figure 15 shows a photo of the EDC experimental setup used to demonstrate the utilization of the proposed CubeSatFI fault injection platform. In addition to the elements already described for the target system (the EDC), the setup also includes the RF generator to emulate the communication with the satellite, a serial/USB convertor, the power supply, and the host computer (the PC) that runs the CubeSatFI and the OpenOCD.

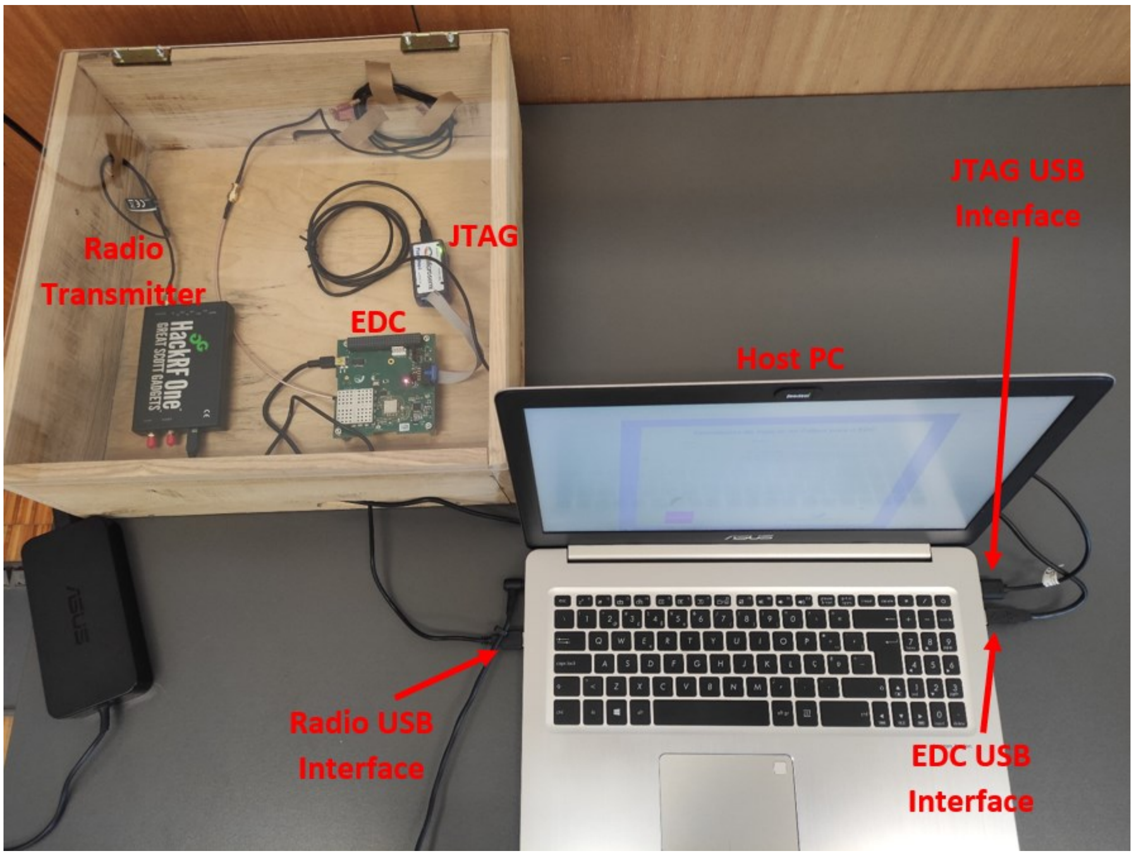


Figure 15 - Photo of the EDC Experimental Setup

Results Analysis

Figure 16 shows the general impact of faults (i.e., failure modes) while EDC decodes messages from the point of view of the satellite onboard computer (OBC). The confidence intervals (shown in the numeric values in each bar of the chart) are calculated for 95% of confidence, using confidence intervals for proportions in binomial distributions (Bernoulli trials) – the formula is presented in Table 1.

$$p \pm z\sqrt{\frac{p(1-p)}{n}} \quad where\ p = \frac{x}{n}$$

$$x \rightarrow Porpotion\ of\ a\ failure\ mode\ ;\ n \rightarrow sample\ size;\ z \rightarrow from\ Z\ table$$

Table 1 – Formula to calculate confidence intervals for proportions in binomial distributions

The classification of failure modes was made based on the results obtained and includes the following failure mode types:

- **No effect**: The fault had no visible impact on the system, which means that EDC continues to work normally, and all the messages are well decoded and sent correctly to the OBC.

- **Blocked**: The system blocks and stops sending decoded messages to the OBC. The only way to remove the target system from this failure mode is through a hard reset.
- **Wrong results**: The system sends messages with wrong information. The target system needs a hard reset after entering this failure mode.
- **Hang & Wrong results**: After the injection, the system hangs for a moment, and after a while starts o sending messages with wrong information. Needs a hard reset to leave this failure mode.

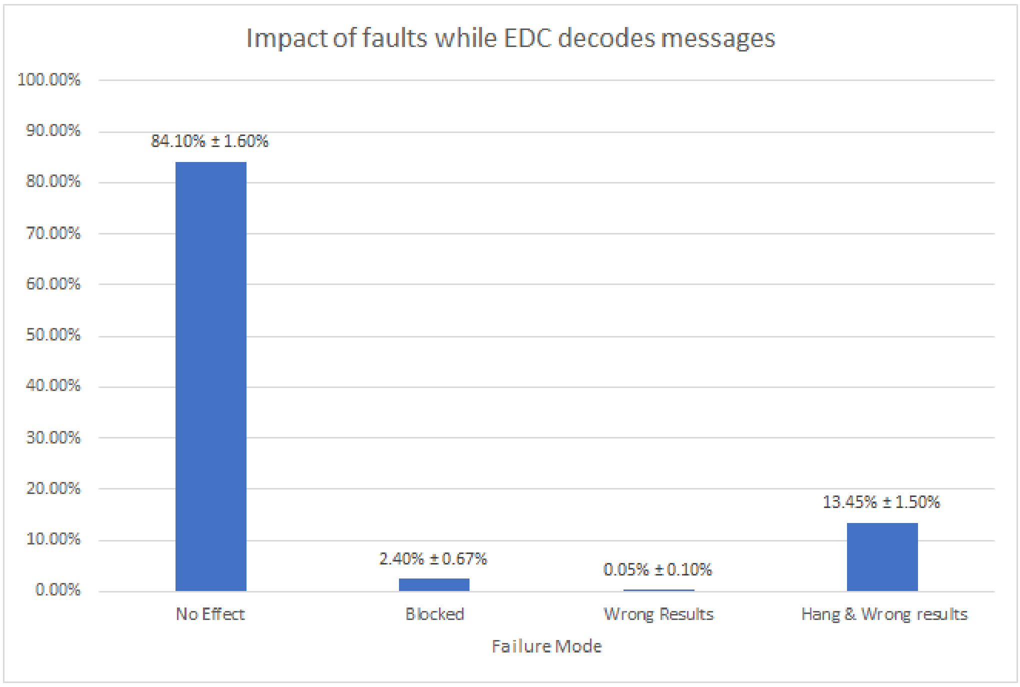


Figure 16 – Impact of faults while EDC decodes messages

The results in Figure 16 show that most of the faults (84.1%) had no effect on the behavior of the software running on the EDC. This result is not surprising if it is considered the inherent redundancy existing in computer systems and in software. Furthermore, this high percentage of “no effect” faults have been consistently observed in many fault injection studies (e.g., [20], [22], [32], [57]), including in a fault injection study done with a COTS-based payload system from a NASA project [9]. The more detailed analysis presented further on (about Figure 17 results) explains (at least in part) this very high percentage of “No impact”.

The analysis of the other failure modes shown in Figure 16 also shows some interesting results. A very small percentage of faults caused wrong results (0.05%). Since the messages have a well-defined format, these wrong results are easy to detect. In other words, failure modes that represent silent data corruption (SDC) were not observed, which consists of erroneous results that are often not possible to detect and do represent a serious risk.

The percentage of faults that crash the EDC software (“Blocked” failure mode) is also quite small (2.4%). In previous fault injection experiments reported in the literature (e.g., in [9]), this type of failure mode appeared in a much higher percentage of faults. One possible reason to explain the low percentage of "Blocked" failure modes in this experiment could be the fact that the EDC runs a very simple real-time operating system (FreeRTOS), since crashes are often caused by operating system crashes.

Still in the results shown in Figure 16, 13.45% of the faults, the EDC shows a strange behavior, starting by not responding at all and then, after some time, starts to send messages with wrong values. A full understanding of this behavior would need a deeper analysis to identify the software idiosyncrasy that originates this behavior (and maybe find a way to make the software more robust in these particular cases). Although a detailed analysis of the results was not carried out in order to try to explain this failure mode, it is worth noting that CubeSatFI records the exact program location affected by the fault (the value of the Program Counter when the fault was injected), allowing the detailed analysis of the fault effect at low levels of the object code.

Figure 17 shows the failure modes observed for the faults injected in each processor register. Two observations are quite evident:

a) faults injected in the registers Program Counter (PC), Stack Pointer (SP), Link Register (LR), and R7 have a strong impact on the EDC software, and
b) faults injected in many general-purpose registers (e.g., R1, R2, R4, R5, R6, R8, R9, R10, R11, R12) have no impact at all.

The PC, SP, and LR are special registers of the processor, therefore it is expected that any fault injected into one of these registers will cause the system to perform an incoherent behavior or even completely block the system. This is not a surprising result.

However, the fact that faults in many general registers have no impact is much more interesting. It means that the software running on the EDC rarely uses those registers or does not use them at all. This could be explained considering the way the C compiler (the EDC software was developed in C language) translates the source code into object code, which tends to use some preferential registers. In Figure 17 we can observe that among general-purpose registers, it seems that the compiler mainly used register R7 (and also a bit R0 and R3), as faults injected in other general-purpose registers had no effect. Obviously, this is highly dependent on the actual source code and also on the compiler optimization switches. This is also the reason why the susceptibility of CubeSat boards to SEU-induced faults is highly dependent on the actual software running on the target system: if the software running on the EDC was more complex or the compiler switches are different, the percentage of "No effect" could drop dramatically.

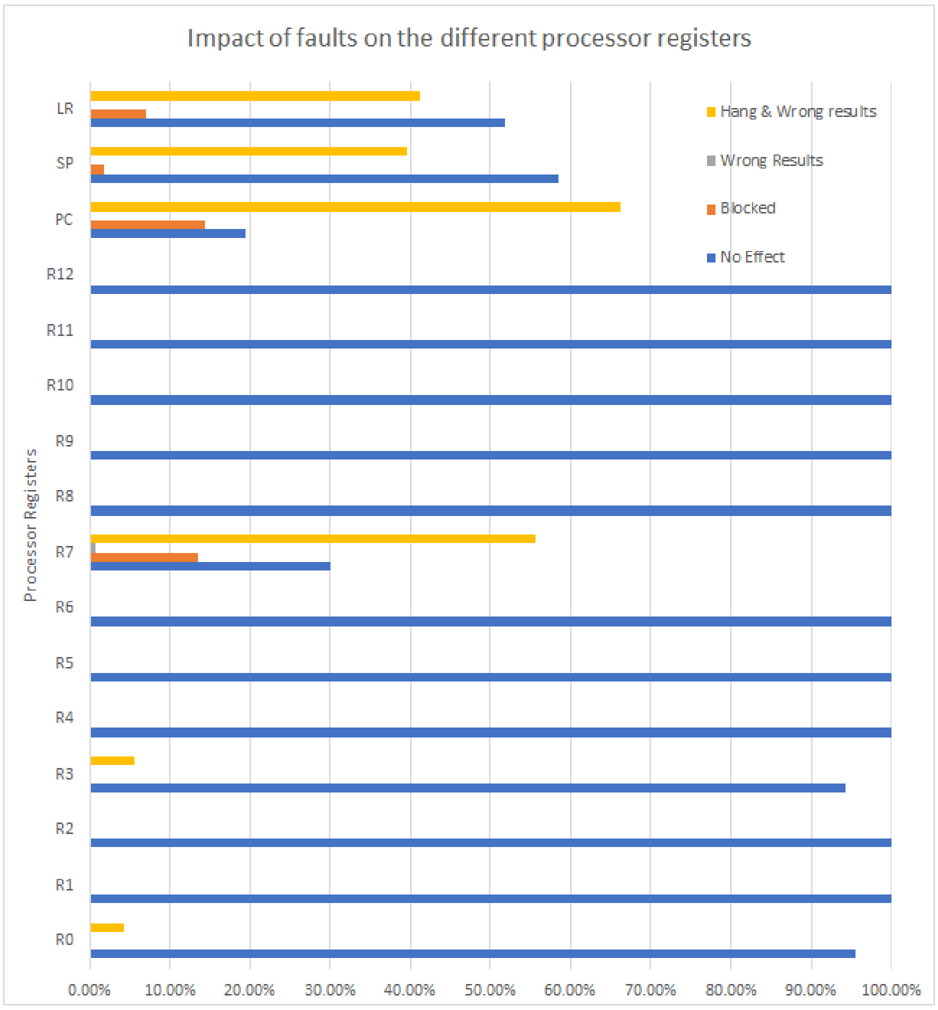


Figure 17 - Impact of faults on the different processor registers

Results shown in Figure 17 provide “food for thought” and give some room for speculations/observations. Clearly, the impact of SEU-induced faults is very dependent on the actual software and Figure 17 shows an important reason for that, which is related to the way the processor resources (especially the registers) are used by the software. This suggests that fault injection campaigns should be executed as a routine procedure during the software development process, as part of the software verification and validation strategy. Even small changes in the software (or in the compiler switches) that lead to a different utilization of the processor registers by the object code could have a considerable impact on the software resilience in the presence of faults caused by space radiation.

Another observation from Figure 17 (related to the fact that the compiler often uses just a few processor general registers) is that we can see the “free” registers as useful resources to develop software fault-tolerant techniques. Given the nature of SEU (i.e., it is caused by a single particle that causes normally a one-bit flip), if software fault tolerance techniques do not use the same registers used by the original software, the effectiveness of such software fault tolerance mechanisms could be much higher.

Figure 18 shows the breakdown of the failure mode results according to different bits affected by the faults. A clear conclusion is that when the faults affect the 16 less significant bits groups ([1-8] and [9 -16]), the impact of faults is much higher than for the other bit positions. On average, when the faults are injected in the 16 less significant bits of the registers, the target system behaves outside the expected behavior (i.e., failure modes showing abnormal behavior) in about 20% of the faults. On the other hand, when the fault was injected in the first 8 most significant bits group ([17-24]), we observed that, on average, 87.55% of the faults have no impact on the target system. A similar result was observed for faults injected in the last 8 most significant bits group ([25-32]), as 91.68% of the faults had no impact on the target system behavior. In short, looking at the faults injected in the EDC system is possible to conclude that faults injected in the less significant bits lead to much more drastic wrong behavior in the system than faults injected in the second half of the 16 most significant bits.

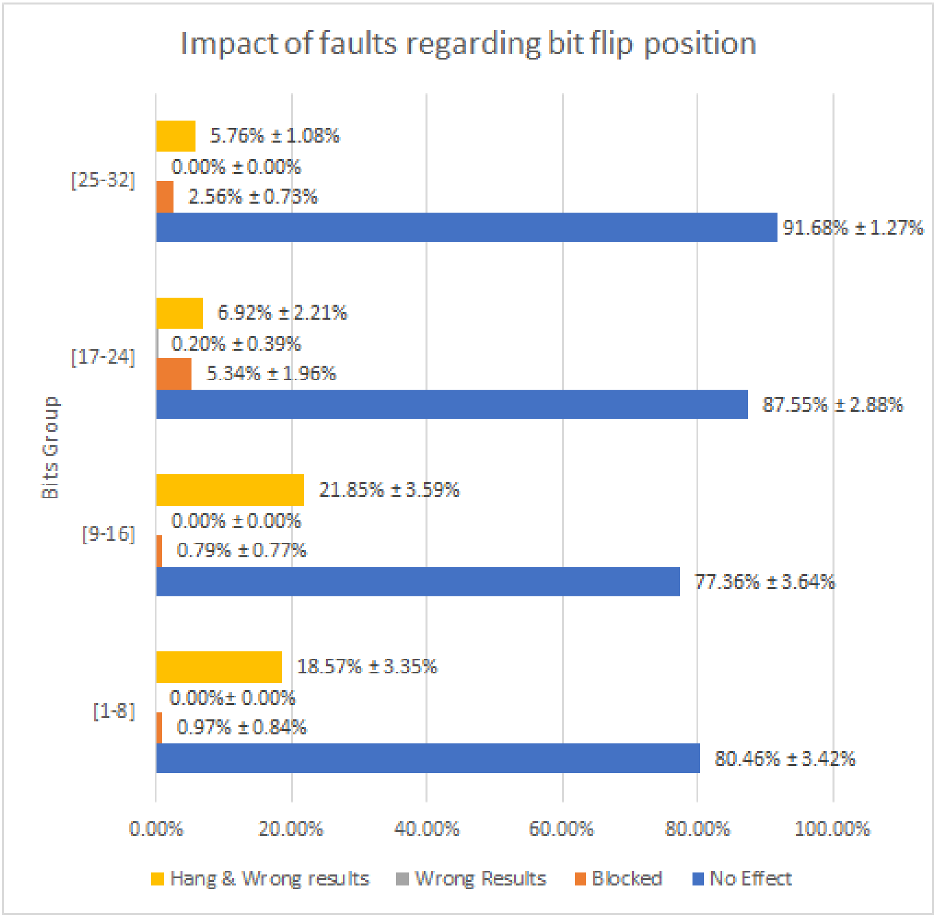


Figure 18 - Impact of faults regarding bit flip position

The fact that the CubeSatFI allows one to choose the precise bits and registers to be affected, opens the possibility to design more focused experiments and, consequently, evaluate in more detail specific erroneous behavior of the software running on the target system. Considering the data collected with the experiment presented before, it was decided to perform a second fault injection campaign of 1000 faults with random time fault triggers, in order to evaluate the EDC software behavior when the faults are injected on the less significant bits of the registers that show high fault impact. Figure 19 shows the results of the faults injected on the less significant bits of the LR, SP, PC, and R7 processor registers and shows that the less significant bits of these registers have a strong negative impact on the EDC behavior in the presence of faults. Faults injected in the less significant bits of the registers PC and R7, on average, cause more than 90% of wrong results, while the registers LR and SP show 68% and 75%, respectively.

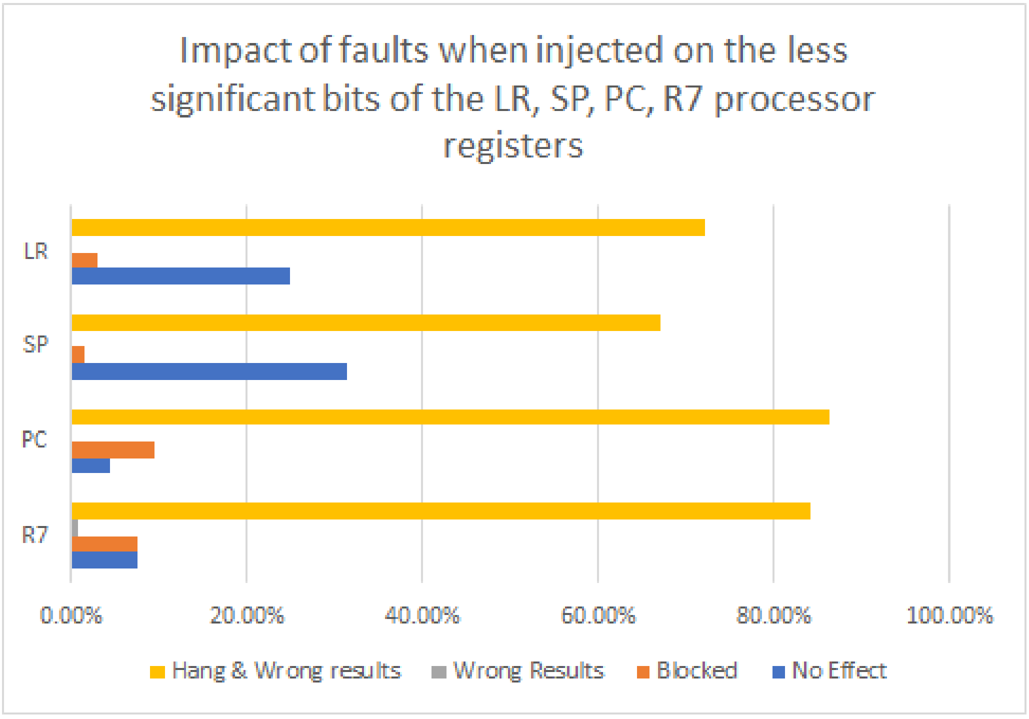


Figure 19 - Impact of faults when injected on the less significant bits of the LR, SP, PC, R7 processor registers

The reported experiment showed that the CubeSatFI can effectively identify weak points of the target system concerning the impact of SEU caused by space radiation. These weak points are related to structural (i.e., hardware) aspects of the target systems, but the reported results also show that CubeSatFI has a great potential to help to improve the resilience of the software running on CubeSats.

4.3 Concluding Remarks

The CubeSatFI functionalities and screens are presented in this chapter. CubeSatFI intends to be a practical tool to design fault injection experiments for CubeSats considering this dual utilization part of the software verification and validation process and as a tool to evaluate the effectiveness of software-implemented fault tolerance techniques.

Hence, it is presented a real use case of fault injection in the Environmental Data Collector (EDC). The results show the failure modes observed and demonstrate the type of fine-grain analysis that can be done with the fault injection results obtained with CubeSatFI, showing the potential of the proposed fault injection platform.

In the next chapter, it is presented an enhanced software development process for CubeSats to cope with space radiation faults that takes advantage of the use of the CubeSatFI.

Chapter 5
Integration of Fault Injection in the Software Development Process

Besides the other objectives already mentioned before, this thesis also aims to propose a set of structured steps (and tools to support such steps) to enhance the classic software development process used in CubeSats, focusing particularly on the Verification and Validation (V&V) phase.

As already explained above, CubeSats are made of COTS components (both standard hardware and software) that are not prepared to deal with the harsh conditions of space missions. In particular, space radiation tends to cause potentially frequent SEUs that originate transient hardware faults in the CubeSats boards. Since existing CubeSats boards are unable to mask or tolerate such faults, the alternative is to add software mechanisms to tolerate the effects of transient faults caused by space radiation.

It is worth mentioning that existing CubeSats boards normally include error detection and correction in the memory and in communication channels, but do not have any other mechanism to avoid or mitigate transient faults affecting the processor. Such faults cannot be tolerated in existing CubeSats, which is an important reason why CubeSats are considered not reliable enough for critical or even important missions.

This thesis proposes the systematic use of software-implemented fault tolerance (SWIFT) mechanisms to tolerate or mitigate the effects of the hardware transient faults due to SEUs. These faults mainly affect the processor (because memory and communication in CubeSat boards are protected by a simple parity bits mechanism) and cause erroneous behavior in the software, which is precisely the problem addressed by SWIFT techniques. Since the impact of transient faults on the software behavior is highly dependent on the software code and data, the proposal is to include fault injection as a systematic and mandatory step of the CubeSats software lifecycle process, which allows a precise and on-demand use of SWIFT.

This proposal will be presented in a scientific paper that will be submitted at the 27th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2022). The current version of the paper can be found in Appendix C - Enhanced software development process for CubeSats to cope with space radiation faults.

This chapter aims to present the proposal of an enhanced software development process for CubeSats.

5.1 Context and Assumptions

The proposed approach uses fault injection as an integral part of the development environment for CubeSats software and includes three high-level steps:

a) Sensitivity evaluation (verification) of the software in the presence of faults caused by space radiation;

b) Enhance the software with targeted software-implemented fault tolerance (SWIFT) mechanisms; and

c) Validation of the effectiveness of the SWIFT mechanisms to confirm that the software is immune to space radiation faults.

These added steps to the V&V process must be carried out during CubeSat software development, as well as every time the CubeSat software has an update, to ensure that the impact of faults caused by space radiation is tolerated by the CubeSat software.

CubeSat boards high-level architecture can be divided into three different layers as shown in Figure 20. The **hardware layer** is the physical part of the CubeSat composed of several components, such as the onboard computer, payload boards, solar panels, RF antennas, among others. The layer above is the **system software** that includes the operating system (e.g., FreeRTOS[11], eCos[12], among others) and, depending on the specific board, may include other software elements such as drivers and software designed to deal with specific sensors or actuators. For example, the system software may also include software developed by the board manufacturer, such as the software that controls the orientation and altitude of the satellite. The CubeSat **application software** aiming to perform different types of tasks is developed to run on top of this system software.

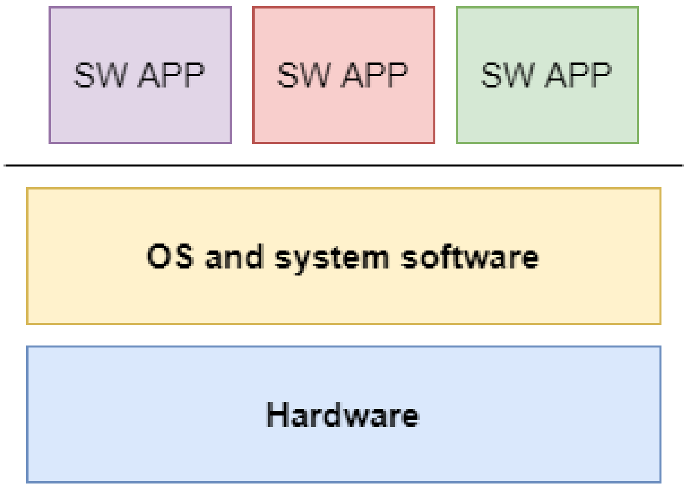


Figure 20 – CubeSat boards high-level architecture

At the **hardware layer**, CubeSats boards use regular COTS components but the boards include several mechanisms to cope with the rough conditions of space missions [15]. Although made of COTS components, CubeSat boards are mechanically ruggedized with layers of resin coating for mechanical and thermal protection. Additionally, memory is protected with error detection and correction bits, as well as communication channels

[11] https://www.freertos.org/RTOS.html (accessed Jun. 21, 2022).
[12] https://www.ecoscentric.com/ecos/index.shtml (accessed Jun. 21, 2022).

are also protected with error detection and correction mechanisms (e.g., CRC and parity checks) provided by the communication protocols and associated hardware of the communication links. Memory, in particular, must be protected against transient bit-flip errors due to space radiation as the memory chips represent a large silicon area exposed to radiation, making SEU in memory very frequent.

Fortunately, the protection of uniform hardware structures such as the memory and the communication channels is very simple and is in fact a common practice in standard COTS hardware boards for all sort of applications. CubeSats simply take advantage of available standard solutions such as extended Hamming codes [16] for single error correction and double error detection in the memory and forward error correction codes [17] used to deal with errors caused by transient faults in communication channels. These mechanisms are well aligned with the CubeSat "philosophy" of low cost, low energy consumption, and low weight.

Protecting the processor from the SEU effects is a much more complex task because the processor is not a regular and simple structure such as the memory. The use of space-grade processors that resist to space radiation is not an option for CubeSats, as the cost of such processors is several orders of magnitude higher than the cost of common COTS processors. The obvious solution would be to adapt classic fault-tolerant architectures with massive levels of redundancy, as the ones used in large-scale satellites [58] or in the aircraft industry [59], [60]. Unfortunately, these well-proven solutions are not an option for CubeSats, even if designed around COTS components, as they are expensive, heavy, and require high power consumption. Classic architectures used in avionics and in large satellites would require pairs of duplicated processors and the inherent hardware logic to compare/vote the results from the different signals, which would ruin the simplicity and low cost of CubeSats.

The current situation is that there are no fault-tolerant CubeSat boards available from manufacturers that solve the problem of transient hardware faults in the processor at the **hardware layer**, as represented in Figure 20.

A recent research work (Ph.D. thesis of C. Fuchs, December 2019 [19])) proposes a novel on-board-computer architecture for very small satellites (<100kg), promising high reliability without using radiation-hardened semiconductors. This proposal uses a combination of hardware and software-implemented fault tolerance techniques. In terms of the high-level architecture shown in Figure 20, the architecture proposed in [19] tolerates transient faults due to SEU using a clever combination of solutions at the **hardware layer** and at the **system software layer**. However, in spite of this promising research result [19], there are no fault-tolerant CubeSats boards commercially available and CubeSats have remained as very low-cost small satellites for non-critical low earth orbit (LEO) missions.

One important advantage of classic fault-tolerant techniques applied at a low architectural level (e.g., triple modular redundancy [49]) is that these techniques provide a reasonable transparent solution for the development of software applications on top of a fault-tolerant architecture. That is, the developer of application software does not need to worry about possible transient faults, as they are tolerated at the lower levels of the hardware layer or by the system software [61], [62].

Since there are no fault-tolerant CubeSats boards currently available (and they are not likely to appear in the near future because of the high cost, energy consumption, and weight imposed by hardware fault tolerance), it means that possible solutions for the transient processor faults due to SEU are not transparent for the developer of software applications for CubeSats. This is obviously a clear assumption for any proposal that attempts to solve the problem of transient processor faults in CubeSats through the use of SWIFT techniques, which also includes the approach proposed in the present thesis. The developer of CubeSat applications must be aware that the application may be affected by transient processor faults and deal with the SWIFT techniques needed to tolerate such faults. Naturally, the development of CubeSat applications will become more complex, as the application software needs to deal with both the functional aspects and the SWIFT techniques, but this is the price to pay to assure the required reliability for CubeSat applications running on simple and low-cost non-fault-tolerant boards.

The development of a software components that implement the skeleton of software fault-tolerant techniques is out of the scope of the work developed under this thesis. However, in the context of a future industrial application of the proposed steps, it will be crucial to have a library of SWIFT methods to be used/adapted to each particular situation, in order to simplify and accelerate the development of CubeSat software capable of tolerating the hardware transient faults caused by space radiation. Of course, those techniques should be tailored to the specific software under development, as mentioned before, but a general skeleton or code (e.g., a voter that compares two inputs and signs if they differ) that can be reused could be made available in the form of reusable components available for the software development teams. This will reduce the time necessary to adopt SWIFT techniques into the code under development, making it easier and cheaper to apply the approach proposed in this thesis.

The application of SWIFT techniques at the software application level to tolerate hardware transient faults caused by SEU, as proposed in the present technique, relies on two assumptions:

a) The system software, and specifically the operating system of the CubeSat board, is operating properly after the transient fault, allowing the correct processing of SWIFT techniques at the application level; or

b) Possible malfunctions (errors) caused by the fault can be detected by the error detection mechanisms available in the CubeSat board, so the board can be restarted to re-establish a correct state to run the SWIFT techniques and tolerate the fault.

This means that in the worst-case scenario (bullet b)) when an error is detected or the system crashes as a consequence of the transient fault, the base layers of the CubeSat (i.e, hardware and operating system) should be able to recover the system to a state from which it can operate properly (forward recovery [43]–[45]). To assure this, a key feature of the hardware layer and the system software layer of the CubeSat board is the **effectiveness of the error detection mechanisms** available on the board.

There is a wide range of error detection mechanisms compatible with the low cost, low energy consumption, and low weight required by CubeSats boards. Unfortunately, most of the CubeSats boards currently available have just a few error detection mechanisms.

As mentioned, all CubeSats boards have error detection of two bits errors and correction of one bit in memory using extended Hamming code [16]. The correction of one-bit error is fully transparent, as it is processed at the hardware level, and in case of detection of errors in two bits (no correction), the error must be handled by the system software (in general, the action is to reset the system as these errors are mostly caused by transient faults due to SEU, and they disappear after reset).

Another very relevant error detection mechanism that also exists in all CubeSat boards is the watchdog timer (WDT) [46] that detects deviations of the correct software behavior that changes its timing features (most frequently, WDT are used to detect crashes). WDT can be controlled (i.e., refreshed periodically) by the system software, which makes the error detection transparent to the application software, or can be periodically refreshed by the application software. Other types of simple error detection mechanisms are associated with the memory management units of the CubeSat board and allow the detection of erroneous memory access behavior (e.g., instruction fetch outside the code segments, read/write in memory areas not available, etc.). More sophisticated (and also more effective) error detection mechanisms such as signature monitoring [63], [64], are in general not available in CubeSat boards.

Given the relevance of the assumptions mentioned above (bullets a) and b)) for the approach proposed in this thesis, the author decided to perform a preliminary experiment to evaluate these assumptions in faulty scenarios. The goal is to evaluate the probability of the CubeSat board (hardware layer and system software) to behave correctly after a fault, in such a way that SWIFT techniques can be applied to tolerate the faults. It is clear that SWIFT techniques can only be applied if the operating system is working properly.

A campaign of 10000 faults, selected at random in the space and time domains (to emulate accurately the random effects of space radiation), was injected into the EDC board from an INPE satellite. For this experiment, the EDC (target system) was not running any real software application. Instead, the EDC was just running the real-time operating system (FreeRTOS) and, a "dummy" task that blinked a LED light and refresh the watchdog timer counter. The idea was to evaluate the impact of faults in the system software (mainly the FreeRTOS), to evaluate whether the operating system is running properly after the fault or not.

The results obtained are presented in Figure 21. The confidence intervals (shown in the numeric values in each bar of the chart) are calculated for 95% of confidence, using confidence intervals for proportions in binomial distributions (Bernoulli trials) – the formula was presented above in Table 1.

The classification of the failure modes was made based on the results obtained and includes the following failure mode types:

- **No Effect/OS OK**: The fault had no visible impact on the system. The operating system continues to work normally as expected.
- **Error detection (WDT)**: The fault crashes the operating system, but the watchdog timer detected this erroneous situation and restart the system. After restarting, the system is working normally again.
- **OS CRASH**: The system crashes after being affected and the watchdog timer cannot detect it.

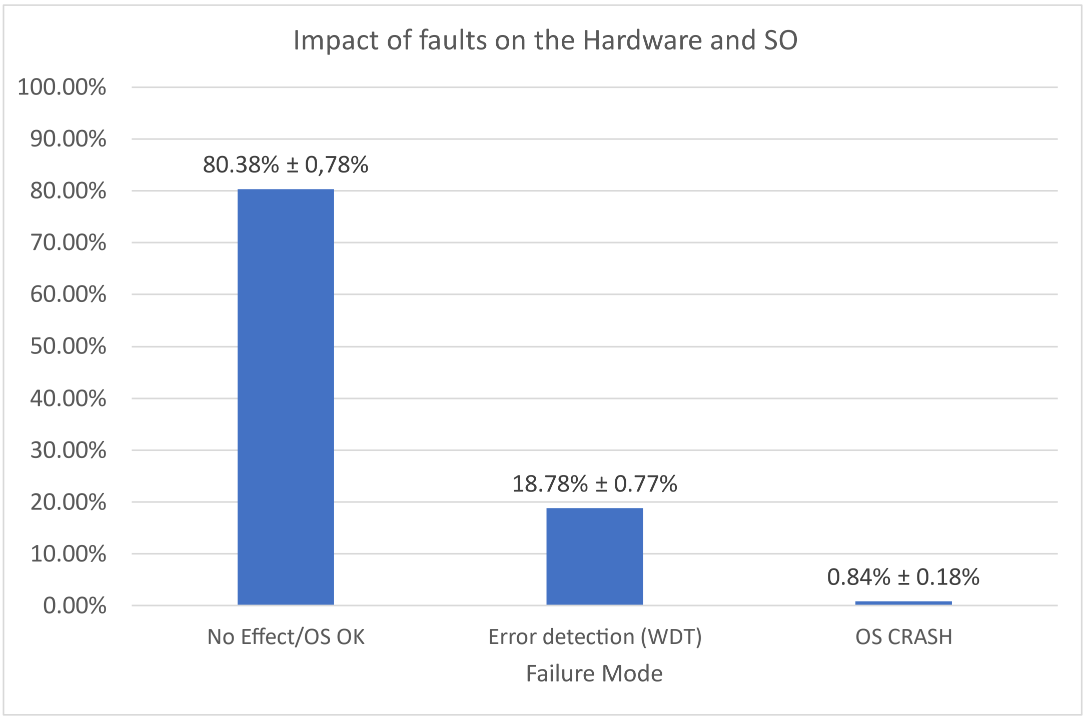


Figure 21 – Impact of faults on the Hardware and SO

The results show that most of the faults did not affect (80.38%) the operating system, which means that the operating system continues operating properly, as expected. Besides, 18.78% of the faults activate the watchdog timer, assuring that after a crash the operating system can restart and back operating properly again. These two values together (**99.16%**) show that the hardware layer and system software layers meet the assumptions described above (both a) and b)) and SWFIT techniques can be effectively applied at the software application layer. This means that software developers can develop applications on top of COTS boards and use SWFIT techniques to tolerate processor transient faults due to SEU at the software application level. It is worth noting that in this experiment the error detection available in the target system (EDC board) was only the WDT. Even so, the percentage of cases observed in which the proposed approach could not work is reduced to **0.84%**. Obviously, the inclusion of additional error detection mechanisms on the board could reduce even further this percentage.

A final word is about the software development approaches used to develop software applications for CubeSats and software assurance practices. In general, CubeSats software applications are developed using the classic waterfall development process

[65] and applying the well-known V-model [65], [66] on top of it. This means that verification and validation (V&V) activities are a common practice in CubeSats software application development, which means that the three additional steps (basically V&V steps) proposed in this thesis will be easily integrated. As already mentioned, the use of SWFIT techniques could be largely facilitated in real projects through the availability of a library of pre-developed SWIFT methods to be reused and adapted to each new CubeSat application.

Regarding software assurance practices, simulation and testing are the most common activities to verify and validate CubeSats software, according to a survey conducted at NASA Ames Research Center [65]. However, even those activities do not receive due attention on CubeSats projects, as an intensive program of verification and validation cannot be accommodated into the limited budget of such projects. Despite this, according to the same survey [65], an emerging trend relies on the use of model-based design methods due to their capability to automate the creation of detailed software design from high-level graphical inputs, and then use automatic code generation to create the code. Although, this type of modelling/software development requires expertise on the part of the developer of the tool used. The definition of requirements for generating the model is fundamental for the fidelity of this model to the desired implementation. Both the correction of bugs found in the tests and the integration of designed modules must be done at a high level, hand-codes are not allowed to guarantee the reliability of the designed model.

The use of rigid verification and validation techniques is not a trend in current CubeSats software development due to the time and budget constraints of such projects. This includes the crucial verification of the possible effects of space radiation-induced faults.

5.2 Enhanced Verification and Validation Steps

The proposal focuses on enhancing the verification and validation of CubeSats software through a set of additional steps. These steps are intended to be the least intrusive possible on the software development life cycle used by the companies, space agencies, and other institutions that are developing CubeSats. Since budget and time are constraints that must be considered, expensive software verification and validation activities are impossible to accommodate in such projects.

The proposed steps require a fault injection tool, but such tools are readily available at low cost, such as the tool CubeSatFI [10], which can be easily integrated into the software development process of CubeSats without significantly increasing the cost of such projects. Moreover, the proposed approach when used in the early stages of the software development life cycle cannot only find weak points caused by space radiation-induced faults, but also can be useful to find software bugs not discovered during integration testing activities. Figure 22 illustrates the proposed additional steps. More specifically, our proposal does not change the previous phases of the existing software development process, but simply adds additional V&V steps after the integration test step, which is always part of the process, no matter the flavor of the software development process used by the CubeSat developer. The proposed steps assume that a fault injector tool such as CubeSatFI [10] capable of injecting transient faults similar to the ones caused by SEU is available:

Step 1 - Evaluate the software sensitivity to space radiation: After integration testing the software is subject to a comprehensive fault injection campaign to evaluate the impact of SEU on the CubeSat behavior. Faults are injected into the processor registers of the target board using a random distribution (both in space - registers- and time), since space radiation tends to affect the processor randomly. This will allow to understand the behavior of the target software in the presence of space radiation that affects the processor of the board where the software is running.

Step 2 - Strengthen the software with tailored software-implemented fault tolerance (SWIFT) techniques: The results obtained in the previous step must be analyzed and the impact of the faults on the target software should be categorized into failure modes. According to the failure modes obtained, it should be decided to add additional SWIFT techniques to the code to avoid failure modes such as **silent data corruption** (erroneous output results with no error detection) or to recover the software after **crash failure modes**. This decision should be taken considering the available resources of the target system and the budget available to implement these techniques. Many SWIFT techniques can be used (see, for example, chapter 5 of [67] or the classic Michael Lyu's book [68]), from simple re-execution and voting to self-checking software. If the target system has enough resources, it is extremely recommended to add SWIFT techniques to increase fault coverage as much as possible. Obviously, that includes additional SWIFT techniques in the CubeSat software after a first version of the software has been through integration testing could be problematic. For fault-masking techniques such as software re-execution and voting [67], [68], the task of adding this technique to existing software is relatively easy. But for other SWIFT techniques such as algorithm-based fault tolerance [68], the existing software must be largely refactored to incorporate the SWIFT technique. With that in mind, adopting these proposed steps in the early stages of software development is quite recommended.

Step 3 - Validate the effectiveness of the SWIFT techniques: After the software is strengthened with additional SWIFT techniques, it must be submitted to regression testing (using a test suite developed in earlier stages of the software development lifecycle) to assure that the functional requirements (and also non-functional requirements such as response time) are still met. The validation of the effectiveness of SWIFT is then performed through a fault injection campaign similar to the one run in step 1. That is, the process enters the cycle proposed in Figure 22 until the desired software resilience in the presence of transient faults is achieved. The objective is to evaluate the effectiveness of SWIFT techniques in the mitigation or toleration of transient faults such as the ones induced by space radiation.

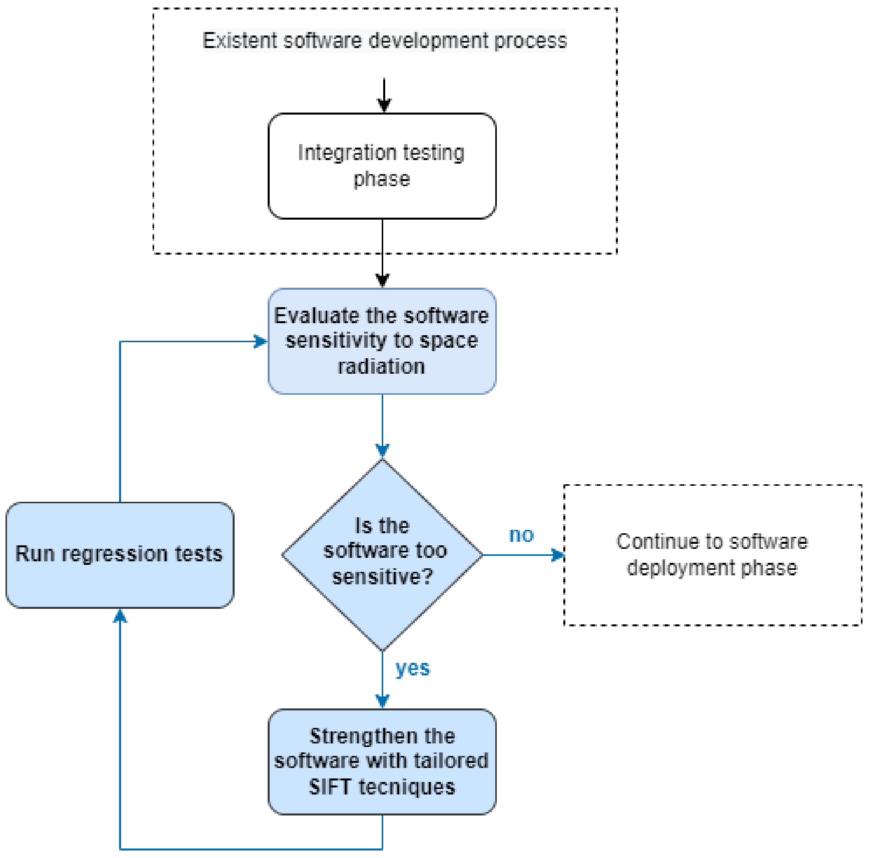

Existent software development process
Integration testing phase
Evaluate the software sensitivity to space radiation
Is the software too sensitive?
no
Continue to software deployment phase
yes
Strengthen the software with tailored SIFT tecniques
Run regression tests

Figure 22 - Enhanced verification and validation steps for CubeSats software development process

The proposed steps should be included in the software development process used in the CubeSat development project. If the project follows the classic V-Model, the proposed steps should be included after the integration testing phase (right-side of the V). If the CubeSat project follows an agile process, the proposed steps should be performed each time that the software has a considerable increment. Since the impact of SEU-induced faults depends on the actual software that is running on the CubeSat, every time the software changes, it is crucial to perform the proposed additional V&V steps. In fact, these steps are quite inline with test-driven development (TDD) used in agile development processes, where the software requirements are converted into test cases and each software increment aims to pass the new set of test cases. After the test pass, the code is refactored, and the test suite is run again to assure that no existing functionality is broken. This cycle is repeated for each new functionality. Similarly, to TDD, when the CubeSat software has a major or even a minor change, the proposed V&V steps should be executed to evaluate the resilience against SEU-induced faults, and the software is considered fully developed when it meets the safety and dependable requirements to tolerate space radiation.

5.3 Concluding Remarks

This chapter presents a set of steps to enhance the software development process for CubeSats to cope with space radiation faults. The proposed solution intends to be applied to the software application developed on top of the COTS components (both hardware and software) made available by the manufacturers. Before the adoption of the proposed steps, an evaluation of the COTS components resilience must be done to assure that after being affected by a fault, these components can continue to operate as expected. The focus of the proposed steps is on the software applications that run on top of these components.

The key idea of the proposed solution is to use fault injection to emulate transient faults caused by space radiation and analyze their impact on CubeSat software. In addition, tailored software-implemented fault tolerance techniques must be added to the software under test, aiming to tolerate or even mitigate the effect of space radiation-induced faults. To finalize, the effectiveness of such techniques must be evaluated with a new fault injection campaign. In short, the proposed steps can be summarized in the following points:

1. Evaluation of the software sensitivity to space radiation;
2. Strengthen the software with tailored software-implemented fault tolerance (SWIFT) techniques;
3. Validate the effectiveness of the SWIFT techniques.

This chapter also discussed the context and assumptions required for the proposed approach, and showed through a simple controlled experiment that such assumptions are easily met in current CubeSat boards, considering that the EDC board from INPE is representative of most CubeSat boards. Is not worth mentioning that an important aspect for the actual application of the proposed approach is the availability of fault injector tools such as CubeSatFI as part of the software development environment to allow the execution of fault injection campaigns in an easy and automatic way (and at low cost).

Chapter 6
SBCDA Use Case and Results

This chapter demonstrates the proposed approach using three different embedded software running in the Environment Data Collector (EDC) CubeSat board, which is part (payload) of a constellation of satellites being developed by the Brazilian National Institute for Space Research (INPE). The following use case provides a realistic insight on the effectiveness of the steps proposed in the previous chapter.

6.1 CubeSat CONASAT-1

The Brazilian Environmental Data Collection System (SBCDA) has operated since 1993 after the launch of the SCD-1 data collection satellite. The SBCDA aims to collect data such as wind speed, rainfall, temperature indices, wildlife observation, among others. In addition to the SCD-1, the system is currently supported by the SCD-2, CBERS-4, and CBERS-4A satellites (much larger than CubeSats), which carry on board an analog transponder that relays the signals from the data collection platforms (DCP) to the receiving stations (RS) on the ground.

The EDC – presented previously in Chapter 5 – is a new payload developed to meet the demand for a signal receiver CubeSat-compatible to provide onboard signal processing. The EDC design uses COTS components which gives it a low-cost, however, it makes the EDC system less reliable than the analog transponder. By offering onboard processing, the EDC is capable of expanding the SBCDA systems service coverage. The development of this payload is convergent with the CONASAT project [69], which aims to launch a constellation of low-cost nanosatellites to renew and expand the Brazilian data collection system.

CONASAT-1 - Figure 23 - is the first satellite in a constellation of low-cost nanosatellites based on COTS components. CONASAT-1 is based on a CubeSat platform with a size of 1U, i.e., this satellite has a cubic shape with edges of 10 centimeters. CONASAT-1 uses the hardware platform developed by the EnduroSat company[13]. The CONASAT project team is responsible for developing the flight software for the onboard computer (OBC), and together with the EDC team, carrying out the integration of the satellite with this payload. In addition to the OBC and the EDC, the CONASAT-1 hardware architecture comprises two UHF antennas, a UHF transceiver, an electrical power system (EPS), and a battery pack. Figure 24 illustrates the hardware architecture in block diagram level described.

[13] https://www.endurosat.com/ (accessed Jun. 24, 2022)

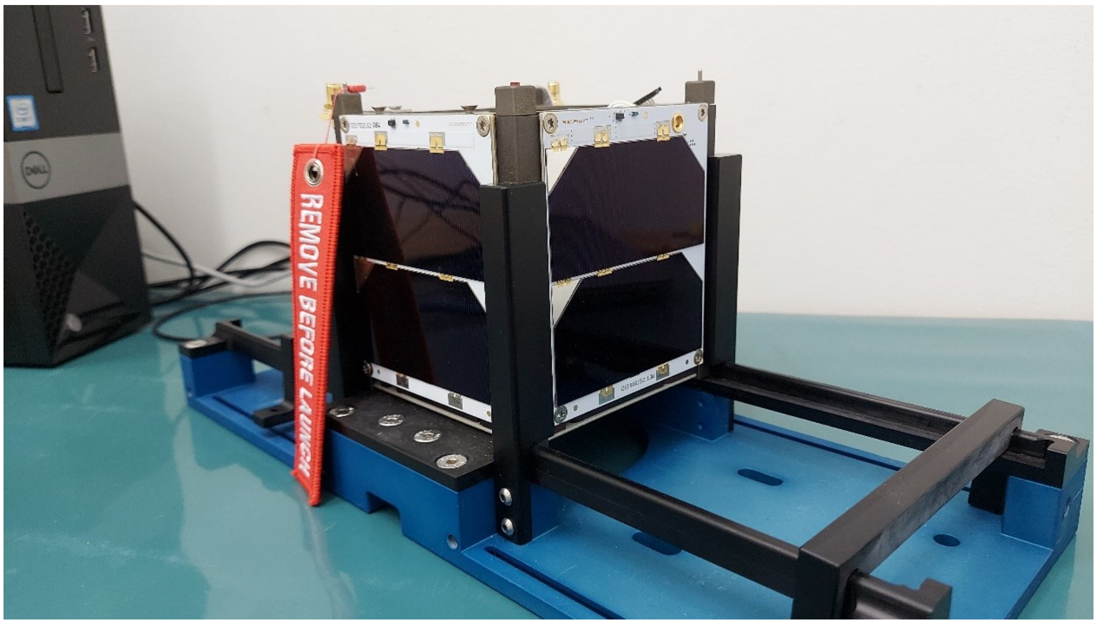


Figure 23 - CONASAT-1 [69]

The UHF antenna - Payload is dedicated to receiving signals from DCPs and is connected to the EDC, while the UHF antenna - TMTC is used for communication with the RSs and is connected to the UHF transceiver. The UHF transceiver is the subsystem responsible for receiving and transmitting the telecommand (TC) and telemetry (TM), respectively. The EPS subsystem supplies power to the entire platform through six solar panels and several voltage converters. The platform also has a battery pack with a capacity of 10.2 Wh. The OBC is responsible for configuring, controlling, and commanding the operation of the subsystems. The telecommands received from an RS are decoded in the OBC to control the subsystems onboard the satellite. The OBC is also responsible for monitoring the overall health of the satellite. A health assessment can be performed in several ways, depending on the subsystem being assessed. Telemetry sensors are used to verify that the parameters of a given subsystem are acceptable (such as temperature or voltage level). Telemetry data collected from each subsystem is also stored for transmission to an RS. The OBC acts as the I2C bus master for transmitting commands to the EPS subsystems, UHF antennas, and UHF transceiver. The flight software implements in the OBC a routine of commands and requests to control the data processed by the EDC. This is performed through a UART communication interface. With the payload data in hand, the OBC uses the USART interface to transfer it to the UHF transceiver. The UHF transceiver transmits through the UHF antenna TMTC at the frequency of 462 MHz. While the beacons are transmitted by the same antenna at the frequency of 435 MHz. The UHF transceiver is configured for a baud rate of 9600 bits per second.

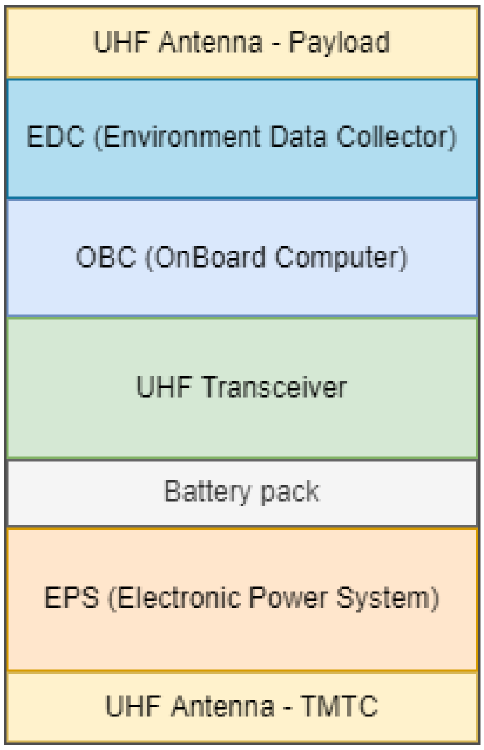


Figure 24 - Block diagram of CONASAT-1 Hardware Architecture Overview

6.2 Application of the Proposed Approach

Aiming to demonstrate the effectiveness of the proposed approach in the software development process for CubeSats, three different embedded software applications have been deployed on the EDC board that will be used on CONASAT-1 CubeSat. The three embedded software applications (for testing purposes only) are:

- **Matrices:** It is a program that computes the result of the multiplication of two matrices 30 times and at the end of each run, calculates a cyclic redundancy check (CRC) for the result of the multiplication. After the 30 runs, calculate a final CRC of the 30 CRCs previously calculated. In this experiment, the program uses two 30x30 integer matrices.
- **PI:** Computes the value of π using the Leibniz formula. In this experiment, the program computed the π using 60000 terms.
- **Fibonacci**: It is a recursive program that computes the sequence of Fibonacci and sums the calculated elements. In this experiment, the program computed the sum of the first 30 elements of the Fibonacci sequence.

It is worth mentioning that these payload software applications do not correspond to the software payload that is going to fly in the future CubeSats from INPE. However, they run on top of the real software running on the EDC board, namely the FreeRTOS operating system and all the software needed for exchanging messages between the EDC board and the OBC board. In practice, the three payload software applications developed have been designed to impose a considerable processing load on the EDC board and reproduce the case of payload software that takes a considerable amount of time to execute.

The fault injection (for the software sensitivity evaluation step) was performed using CubeSatFI, the fault injection tool presented in this thesis that takes advantage of the modern features of the actual microcontrollers by injecting faults in a fully automated way through the JTAG interface.

The target of the injected faults is the registers of the processor of the EDC board. As discussed before (in Chapter 2), the processor is the main weak point for the reliability of CubeSats boards in the presence of space radiation. In the ODC board, memory is protected with single error correction and double error detection parity codes, and the message exchange between the EDC and OBC boards is also protected with error detection mechanisms.

The fault injection campaigns consist of 2000[14] faults. All faults are single bit-flips faults, as this model is broadly accepted as a realistic simulation of SEU faults. With that in mind, the register affected by each fault was selected randomly (among all the processor registers) and the bit of the register affected by the faults was also selected at random. The trigger of each fault is also randomly defined within an injection window (i.e., within a time interval defined by the tester). The injection window interval was defined as between 2 and 4 seconds after resetting the CubeSat to assure that each fault is injected into the system without having the effects of previous faults.

At the end of the injection window, the target system will be running for a period of 26 seconds to collect data for further analysis of the effects of the fault. The results produced by each software application are sent to the host computer that is executing the CubeSatFI through the UART interface of the EDC payload board. The results of the campaigns are saved in a file to further analysis.

These campaigns with randomly injected faults (both in the register space and in time) are appropriated to emulate the effects of transient faults caused by SEU, as space radiation tends to affect the processor in a random way. It was decided to keep the single bit-flip model and not to include faults injected in multiple bits of registers because these multiple bits faults (caused by space radiation bursts) tend to cause a drastic impact on the software and are easy to detect, and consequently are easy to handle.

It is worth noting that due to the random nature of the injection process, the injected faults may affect either the payload software application or the EDC software, namely the FreeRTOS operating system and the software used for exchanging messages between EDC board and the OBC board, which represents a realistic scenario for SEU faults. However, the EDC board and the existing software are quite resistant to SEU faults, as shown previously in Figure 21.

[14] More faults are being injected to increase the confidence of the results presented. Those results will be presented in a scientific article. Due to time constraints, they are not presented in this thesis.

The process was applied following the three additional V&V steps of the proposed approach:

1. Sensitivity evaluation by applying the fault injection campaigns (one campaign for each payload software) to the original software (i.e., without specific SWIFT techniques, unless some error detection techniques, such as watchdog timer, that are available in the EDC board).
2. Strengthen of the payload software with a simple SWIFT technique that consists of r-execution of the payload software and voting of the results.
3. Validation of the effectiveness of the SWIFT technique through the fault injection campaigns.

The next subsection presents and discusses the results.

6.3 Results Discussion

Figure 25 shows the general impact of faults in the three scenarios of running the payload applications on the EDC board, corresponding to step 1, before the implementation of SWIFT techniques and considering the 2000 faults injected in each case.

Based on the results obtained from the experiment the failures were classified according to the following failure modes:

- **No effect**: The fault had no visible impact on the system, which means that the CubeSat continues to work normally, and the expected results are received by the onboard computer.
- **Silent data corruption (SDC)**: The fault had no visible impact on the system. However, the results sent are incorrect.
- **System crash and restart (WDT)**: The system crashes and the watchdog timer (WDT) is activated. After the WDT activation, the system is restarted and goes back to working properly.
- **System crash and do not restart**: The system crashes, remaining in that failure mode without WDT activation.
- **Erratic behavior**: The system sends wrong information repeatedly that can be detected by the mechanisms available in the CubeSat.

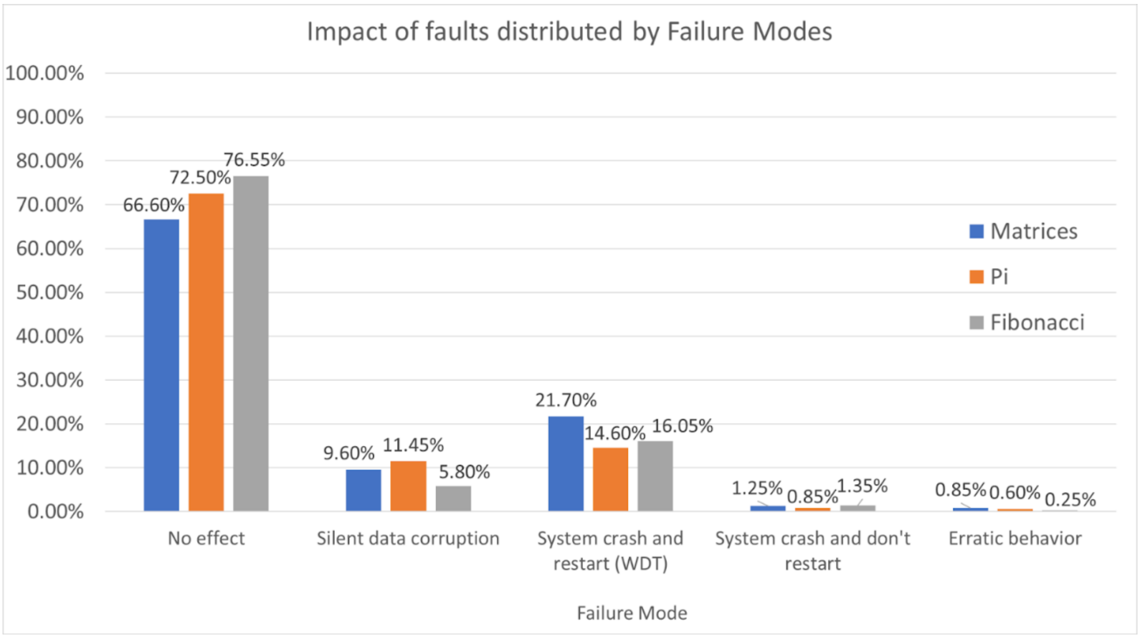


Figure 25 - Impact of faults distributed by failure modes

Considering the failure modes presented above, “silent data corruption (SDC)” is the worst of all and represents a serious risk since the faults that lead to this failure mode are impossible to detect. The code control flow is not affected, instead, the system produces an erroneous result impossible to be detected, which means that SWIFT techniques must be considered to avoid the serious failures caused by these faults. In contrast, when the system crashes and it is detected by the WDT, the system is restarted and back again to work as expected, ensuring system reliability. It is worth mentioning that CubeSat boards are in fact cyber-physical systems and, in real scenarios the CubeSat software applications do not have a comprehensive state that needs to be recovered, such as the case of data-oriented (e.g., database) applications. This is the reason why forward recovery after reset is in general adequate to resume the correct operation.

The “system crash and do not restart” failure mode can be easily managed with external mechanisms such as a Heartbeat system that must receive a signal periodically from the CubeSat payload system. If the signal is not received means that the system is in a blocked situation after a crash and must be forced to restart. Faults that lead to an “erratic behavior” failure mode can also be easily detected by the onboard computer that should restart the EDC board. This is due to the communication protocol between the EDC and the onboard computer of the CubeSat.

The high percentage of faults that have no impact (“No effect” failure mode) in three payload application scenarios is quite normal and corroborates previous fault injection experiments reported in the literature (e.g., [9]). This percentage varies between 66% and 76% depending on the software under test. The reason for such results can be justified by the intrinsic redundancy existing in computer systems and software.

Analyzing in more detail the failure mode distribution for the different processor registers, Figure 26 shows that some processor registers are not affected at all by the injected faults. The reason is that these registers (e.g., R5, R6, R8, R9, R10, R11, R12) are not used by the code. Of course, these situations vary according to the actual software that is being executed in the CubeSat. Furthermore, the way the software uses the available resources of the processor is defined by the C compiler switches during the compilation phase (the EDC firmware and the codes used in this experiment are developed in the C language). In fact, the result of fault injection campaigns can be quite different if the code is compiled with different compilation switches, as this can influence the behavior and performance of the software in execution. Also, the fact that a big number of registers are normally not used by the compiler opens some possibilities to implement extra SWIFT mechanisms.

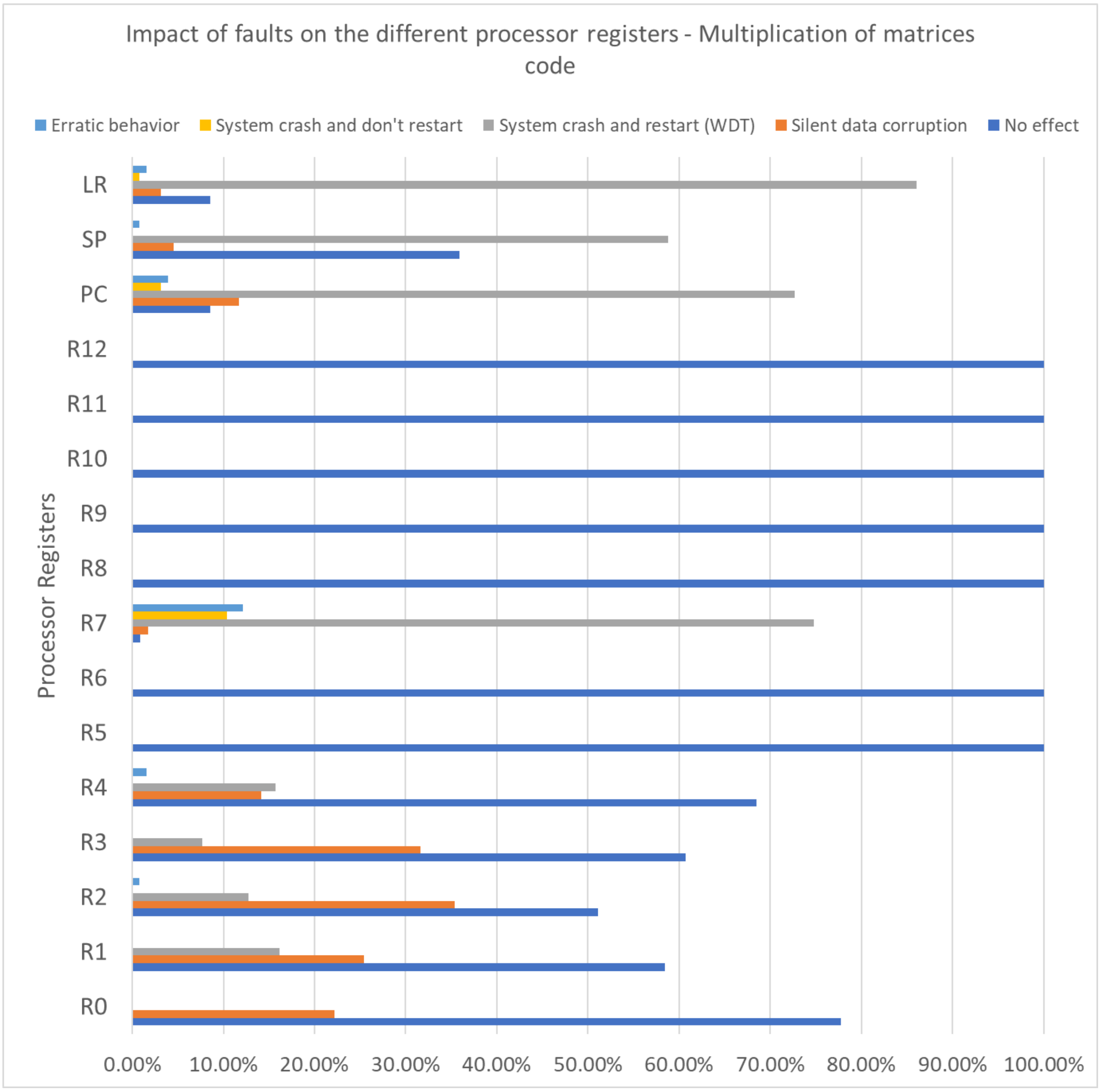


Figure 26 - Impact of faults on the different processor registers - Multiplication of matrices code

To minimize the impact of the transient faults caused by space radiation, the "plain-vanilla" version of the SWIFT technique known as re-execution and voting [67], [68] was added to the three payload software. In practice, the code is executed twice, and the result is voted in order to decide if it is trustable or not. If the two results differ, the code is re-executed and compared with the two previous results. In the end, if the third run does not match either of the previous two, a message of error is sent to the output. In contrast, if the result matches one of the first two, it means that one execution was affected by the space radiation, but the others can be considered trustable.

Figure 27 shows the distribution of the faults according to the different failure modes in each processor register after the application of the software fault tolerance technique explained above. **The results show that the re-execution and the voting can almost eliminate the impact of faults that cause SDC** on the general registers of the processor that are being used by the code (e.g., R0, R1, R2, R3, R4, R7). On R0 and R7 the silent data corruption is totally tolerated, turning these registers immune to this type of failure mode. Also, in the R0, the simple technique of re-executing and voting turns this register immune to space radiation as this register presents a percentage of 100% of “No effect”. Positively, in the other registers the percentage of “No effect” exceeds 80%, ensuring the reliability of the CubeSat against space radiation.

Also, in Figure 27 we can see that the simple software fault tolerance technique used in the multiplication of matrices code just mitigates the effect of faults that leads to SDC on the special registers of the processor (e.g., PC, SP, LR). Looking at the other failure modes, it is possible to see that the results did not change too much. This phenomenon is expected since these registers are special registers of the processor, which means that any fault injected into one of these registers can lead the system to an incoherent behavior or even block the entire system. To strengthen these registers and increase the percentage of “No effect”, more sophisticated error detection mechanisms must be added to the software under development (e.g., self-checking routines).

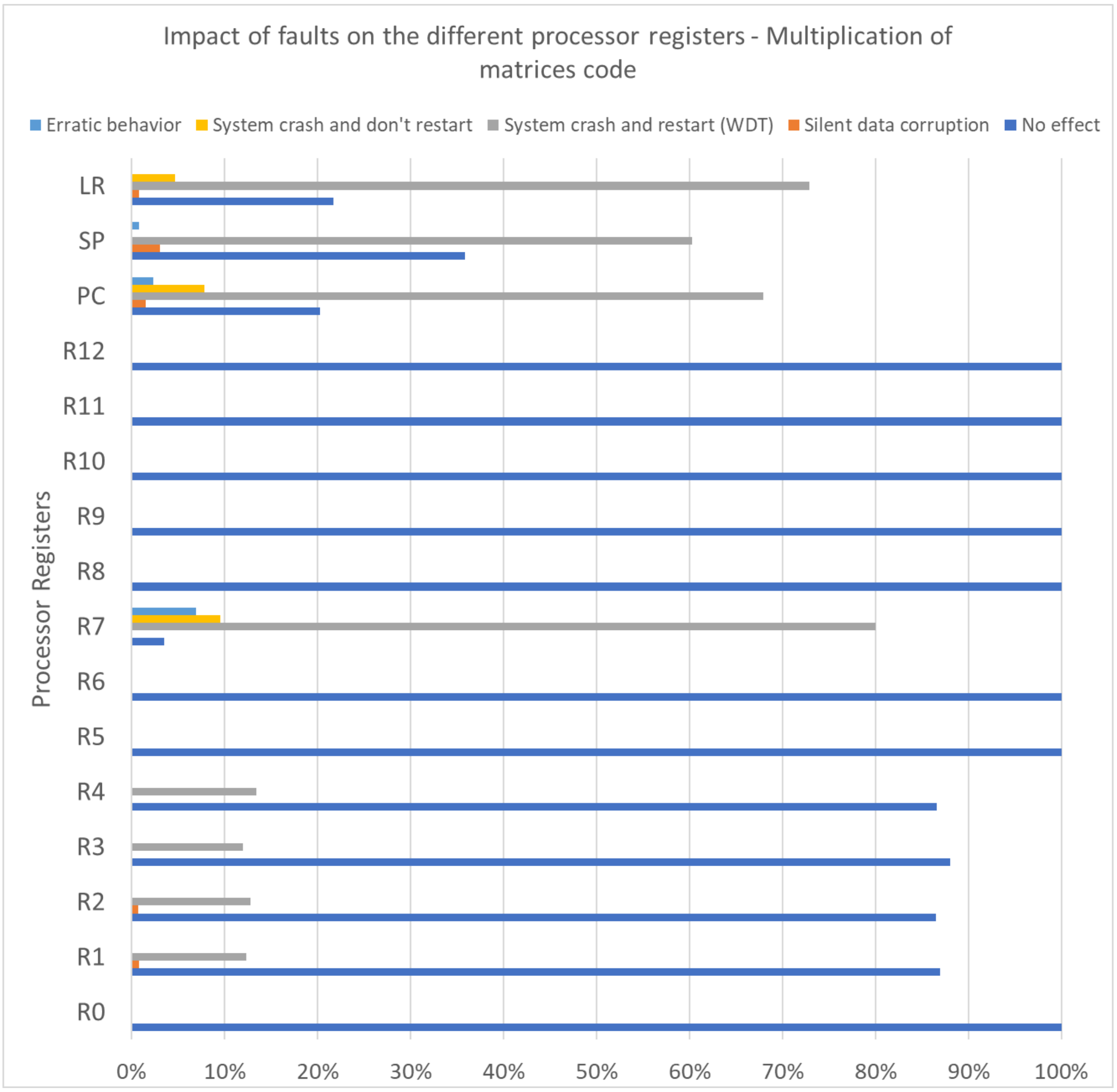


Figure 27 - Impact of faults on the different processor registers - Multiplication of matrices code

Figure 28 compares the impact of faults considering the three application scenarios before and after the payload applications have been strengthened with the re-execution and voting SWIFT technique. Additionally, the analysis considers only the faults that affected the registers that are being used by the software under test (since faults injected in registers that are not used always lead to "No effect" failure mode). As mentioned before, the impact of transient faults is dependent on the actual software that runs on top of the CubeSat and looking at the two graphics in Figure 28, it is possible to see that the sensibility to space radiation varies according to the software running on the EDC board. The multiplication of matrices is the most sensible code, presenting a percentage of "No effect" that almost reaches 42% without any software fault tolerance technique and 59% with a simple re-execution and voting. However, the simple software fault tolerance technique increases the resistance of all the software and in the particular case of the multiplication matrices code, increases the resistance to "silent data corruption" by more than 17%.

Focusing on the effectiveness of the re-execution and voting implemented on the three software, it is concluded that even the simplest software fault tolerance technique can increase the reliability of the CubeSat. The results presented in Figure 28 show that the "silent data corruption" failure mode becomes residual, after the introduction of the re-execution and voting, on all the software under test. In fact, the percentage of "silent data corruption" on the software that does the calculation of the PI is 0%, proving that such software is immune to faults that lead to this type of failure mode. Nevertheless, in the other two codes, we still have some occurrences of SDC, since the technique applied is two simple and even the voter can be affected. However, with a more sophisticated technique (e.g., duplicated voter) the results can be even better.

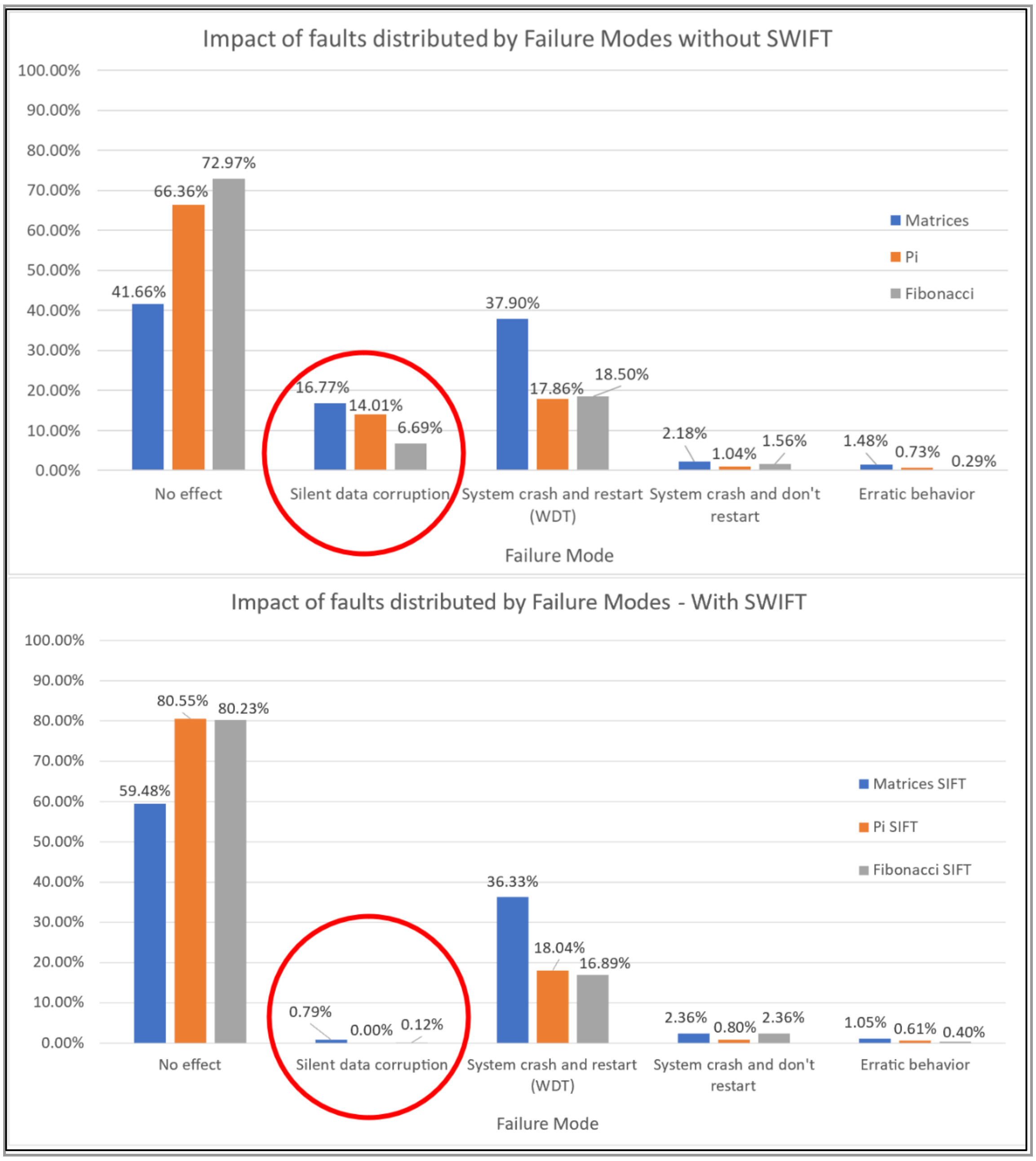


Figure 28 - Comparison of the impact of faults distributed by failure modes on all the software tested before and after being strengthened with SWIFT techniques

In addition to the software fault tolerance technique added to the software, the EDC board includes a watchdog timer (WDT). This error detection mechanism is present in all CubeSats boards and plays a very important role in detecting system crashes. Looking at Figure 28, it is possible to observe that most failures that lead to system crashes are detected by the watchdog timer (i.e., the failures are classified as “System crash and restart (WDT)”). This means that after the system crash is detected by the WDT, the system is restarted and back to work as expected, assuring the availability of the system. Taking advantage of the WDT together with the software fault tolerance technique added to the embedded software, it is possible to make the CubeSats almost immune to SEU faults.

6.4 Concluding Remarks

This chapter presents a use case of the application of the proposed approach using the Environmental Data Collector (EDC), a CubeSat payload board for the Brazilian Environmental Data Collection System (SBCDA) that will be used on all CubeSats from the CONASAT-project. Three embedded software were deployed on the EDC board and submitted to an intensive fault injection campaign aiming to evaluate their sensibility to space radiation.

As expected, the results show that the impact caused by the faults is different according to the software under test. After the distribution of the impact of the faults according to different failure modes, it was possible to observe that all the software under test present a high percentage of "silent data corruption", which represents the worst failure mode caused by space radiation-induced faults, as the fault had no apparent impact on the system but, the results produced are wrong. However, after applying a SWIFT re-execution and voting technique, the occurrence of this failure mode drops to residual values. In fact, in one of the payload applications tested, this failure mode is totally avoided.

This use case, testify the effectiveness of the proposed steps, as well as of the error detection mechanisms and SWIFT techniques.

Chapter 7
Conclusions and Future Work

With all the work presented above in this thesis, it is necessary to carry out a balance and a retrospective of the work accomplished. Therefore, this chapter aims to present these points. In addition, some points of future work that can be carried out are presented.

7.1 Conclusions

This thesis proposes CubeSatFI, a fault injection platform for CubeSat satellites. These satellites use commercial off-the-shelf (COTS) hardware components, which are susceptible to Single Event Upsets (SEU) caused by space radiation. The high rate of hardware transient faults in CubeSats represents an important risk. CubeSatFI allows the easy definition of fault injection campaigns that emulate SEU-induced faults in CubeSat boards, providing effective means to carefully identify the impact of SEU faults and identify possible weak points in CubeSat software.

Although fault injection is a widely used technique in several industrial application areas, including in the space domain, the concrete application of fault injection in the CubeSat industry requires a new perspective and leads to new ways of using fault injection in the development of CubeSats and, more specifically, in the software verification and validation phases. The big difference is that while transient hardware faults are relatively rare in other critical sectors (e.g., automotive, railway, medical devices, etc.), or even in large satellites that use highly expensive radiation-hardened semiconductors, in CubeSat boards, the transient hardware faults due to SEU are very frequent. This means that **processor transient hardware faults should be considered as a "normal" input** due to the fact the COTS semiconductors (particularly the processor) are not prepared to cope with space radiation.

A key idea is grounded on the fact that the impact of transient hardware faults in computer systems is highly dependent on the actual code running on such systems. When the code changes, the impact of faults could change drastically. In the case of CubeSats development, this means that the evaluation of the impact of SEU-induced faults must be carried out every time the CubeSat software has a major change or even a minor update. In other words, the goal is not to protect the hardware of the CubeSat boards (that would be very expensive, as discussed before) but **the real goal is to make the software running on CubeSat boards capable of tolerating processor transient faults** due to SEU. This way, the use of fault injection (as a testing technique) must be a mandatory step in the development of software for CubeSats to evaluate the sensitivity of the software to the effects of processor transient faults, as well as evaluate the effectiveness of the software techniques used to tolerate the faults.

Another key idea is that the negative impact of SEU-induced faults in CubeSats boards should be mainly mitigated or even tolerated by software-implemented fault tolerance techniques **applied at the software application level**. Although this approach has the disadvantage of imposing an extra task to the CubeSats developer of software applications, as it is necessary to take care of the functional aspects of the software and also implement the most adequate SWIFT technique to tolerate processor transient faults due to space radiation, it has the great advantage of being immediately available to be used with the existing CubeSat boards and, above all, it does not have any negative impact on the cost and weight of the CubeSat boards, nor significant impact on the energy consumption.

The use of SWIFT techniques at the CubeSat application level represents a second and very important reason to use fault injection as a key approach to evaluate the effectiveness of such software techniques designed to make CubeSat software more resilient to SEU. If those software techniques are meant to tolerate processor transient faults, the most effective way to test them (and evaluate their effectiveness) is simply injecting such faults.

With that in mind, this thesis proposes an enhanced software development process for CubeSats to cope with space radiation faults. In short, the proposed solution can be summarized in the following steps:

1. Evaluation of the software sensitivity to space radiation;
2. Strengthen the software with tailored software-implemented fault tolerance (SIFT) techniques;
3. Validate the effectiveness of the SIFT techniques.

The proposed solution intends to be easy to adopt in the software development life cycle used by companies, space agencies, and other institutions that are developing CubeSats. The use of fault injection (using available fault injectors, such as CubeSatFI) is a very effective approach to categorize the failure modes caused by transient faults due to SEU, allowing the measurement of the expected (probability of) occurrence of dangerous failure modes such as “silent data corruption”. To mitigate or even tolerate critical failure modes, tailored software-implemented fault tolerance (SWIFT) techniques must be added to the CubeSat software under test. Hence, the software must again be submitted to a fault injection campaign aiming to evaluate the effectiveness of the SWIFT techniques. Following these steps, regression tests should be run to assure that the software functionalities are still working as expected. As mentioned before, these added steps must be performed every time the software has an update, or even a minor change, to evaluate the CubeSat resistance capability against space radiation.

Aiming to demonstrate the effectiveness of the proposed solution, this thesis also presents a use case of the application of the proposed enhanced verification and validation steps proposed using the Environmental Data Collector (EDC), a Cubesat payload board for the Brazilian Environmental Data Collection System (SBCDA) that will be used on all CubeSats from the CONASAT-project.

Results show that the impact caused by the faults is different according to the software under test (which is normal, as the error propagation phenomena and the translation of the erroneous behavior caused by faults into critical failure modes depend on the intrinsic characteristics of the code). Besides, all software tested presents a considerably high percentage of "silent data corruption", which represents the worst failure mode caused by space radiation-induced faults, as the fault had no apparent impact on the system but, the results produced are wrong. By applying a SWIFT re-execution and voting technique, we can reduce the occurrence of this failure mode to residual values. In fact, in one of the payload applications tested, this failure mode is totally avoided, turning the CubeSat immune to space radiation for this critical failure mode.

To conclude, detection fault mechanisms (e.g, watchdog timers, and others) and SWIFT techniques can dramatically increase CubeSats reliability without requiring any change in the current CubeSats boards, making the proposed enhanced software development process (as well as the CubeSatFI) a promising approach to the development of reliable solutions for CubeSats missions.

7.2 Future Work

Nowadays, the interest in the development and deployment of CubeSats is a clear trend in the space industry. Therefore, increasing the reliability of such satellites is a key concern to increase the lifetime and resistance of CubeSats against space radiation. This thesis and the work developed under it, address this problem with a fault injector and a set of steps to enhance the traditional software development lifecycle in CubeSats. Despite this, a set of future work directions can be addressed to further improve the research results already achieved.

First, and the obvious next step is to apply the developed fault injection tool and proposed approach to the onboard computer of the CONASAT-1 satellites. This is dependent on the timing of the CONASAT-1 project and is planned as the next step in the context of the ADVANCED project (which ends in 2024), as soon as the onboard computer board and the software are available testing.

Second, the CubeSatFI fulfills the INPE needs right now, through the emulation of single event upsets caused by space radiation into the registers of the processor according to time and location triggers. Despite this, in the future, the tool can be expanded to include new fault triggers, new fault types, and new target systems. In fact, CubeSatFI architecture is prepared to accommodate such expansion.

Third, the proposed enhanced steps require the implementation of software-implemented fault tolerance techniques on the software application that run on top of the hardware and software made available by CubeSat boards manufacturers. The use case presented in Chapter 6 shows the effectiveness of one of those techniques after being implemented in three different embedded payload software. However, an industrial application of the proposed steps will require the development of a software component that implements the skeleton of the most common software-implemented fault tolerance techniques to keep the effort of implementation less as possible. Is not worth mentioning, that the development of this component is completely out of the context of this thesis. In fact, an extended version of the proposed steps including the software component that implements the techniques mentioned above can be addressed in the future.

Finally, the application of the approach proposed in this thesis should be the starting point to fully redefine the software development approaches used in the CubeSat industry. In fact, although the approach proposed in this thesis can be easily adapted/integrated into the software development methods currently used by CubeSats software developers (and actually this thesis proposes such integration), the author anticipates that the need to make CubeSat software truly fault-tolerant (because faults are very common in CubeSats) should lead to a totally new software development approach for CubeSats. Key elements such as fault injector tools and libraries of reusable SWIFT components should be considered as integral elements of the software development process from the first steps. Additionally, system software (particularly the operating system) of CubeSat boards should be equipped with additional error detection methods and SWIFT techniques, in order to provide a fault-resilient platform for the development of CubeSat applications.

References

[1] California Polytechnic State University, "CubeSat Design Specification (1U – 12U), REV 14, CP-CDS-R14," 2014.

[2] ISO, *ISO 21980:2020, "Space systems — Evaluation of radiation effects on Commercial-Off-The-Shelf (COTS) parts for use on low-orbit satellites."* ISO/TC 20/SC 14 Space systems and operations, first edition, 2020.

[3] S. Shimhanda and T. Murase, "On-orbit Measurements of Radiation Effects on Commercial-Off-The- Shelf (COTS) Hardware for Small Satellites." Jan. 2019.

[4] R. G. Alia *et al.*, "Simplified SEE Sensitivity Screening for COTS Components in Space," *IEEE Transactions on Nuclear Science*, vol. 64, no. 2, pp. 882–890, Feb. 2017, doi: 10.1109/TNS.2017.2653863.

[5] D. Sinclair and J. Dyer, "SSC13-IV-3 Radiation Effects and COTS Parts in SmallSats," *Proceedings of the 2013 Small Satellite Conference*, 2013.

[6] R. Ecoffet, "Spacecraft Anomalies Associated with Radiation Effects," *RADECS 2013 Short Course Proceedings, Chap. VIII*, 2013.

[7] M. Langer and J. Bouwmeester, "Reliability of CubeSats – Statistical Data, Developers' Beliefs and the Way Forward," *Proceedings of the AIAA/USU Conference on Small Satellites, SSC16-X-2*, Jun. 2016.

[8] K. P. Queiroz, S. M. Dias, J. M. Duarte, and M. M. Carvalho, "Uma solução para o sistema Brasileiro de coleta de dados Ambientais baseada em nanossatélites," *HOLOS*, vol. 7, no. 0, pp. 132–142, Dec. 2018, doi: 10.15628/holos.2018.6307.

[9] H. Madeira, R. R. Some, F. Moreira, D. Costa, and D. Rennels, "Experimental evaluation of a COTS system for space applications," *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pp. 325–330, 2002, doi: 10.1109/DSN.2002.1028916.

[10] D. Paiva, J. M. Duarte, R. Lima, M. Carvalho, F. Mattiello-Francisco, and H. Madeira, "Fault injection platform for affordable verification and validation of CubeSats software," in *2021 10th Latin-American Symposium on Dependable Computing (LADC)*, 2021, pp. 1–11. doi: 10.1109/LADC53747.2021.9672584.

[11] T. Zednicek, "Commercial versus COTS+ versus Qualified Passive Components in Space Applications," *ESA Space Passive Component Days*, Jan. 2016.

[12] A. Avižienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan. 2004, doi: 10.1109/TDSC.2004.2.

[13] C. M. Fuchs, N. M. Murillo, A. Plaat, E. van der Kouwe, D. Harsono, and T. P. Stefanov, "Fault-Tolerant Nanosatellite Computing on a Budget," *2018 18th*

European Conference on Radiation and Its Effects on Components and Systems, RADECS 2018, Sep. 2018, doi: 10.1109/RADECS45761.2018.9328685.

[14] T. Wilfredo, “Software Fault Tolerance: A Tutorial,” NASA Langley Technical Report Server, 2000.

[15] F. Davoli, C. Kourogiorgas, M. Marchese, A. Panagopoulos, and F. Patrone, “Small satellites and CubeSats: survey of structures, architectures, and protocols,” *Int. Journal of Satellite Communications and Networking*, Sep. 2018.

[16] T. K. Moon, *Error Correction Coding: Mathematical Methods and Algorithms*. John Wiley and Sons, 2005. doi: 10.1002/0471739219.

[17] Z. Yuan and X. Zhao, “Introduction of forward error correction and its application,” *2012 2nd International Conference on Consumer Electronics, Communications and Networks, CECNet 2012 - Proceedings*, pp. 3288–3291, 2012, doi: 10.1109/CECNET.2012.6201904.

[18] D. J. Sorin, *Fault Tolerant Computer Architecture*. Morgan and Claypool Publishers, 2009.

[19] C. M. Fuchs, “Fault-tolerant satellite computing with modern semiconductors,” Ph.D. dissertation, Leiden University, 2019.

[20] J. Carreira, H. Madeira, and J. G. Silva, “Xception: A technique for the experimental evaluation of dependability in modern computers,” *IEEE Transactions on Software Engineering*, vol. 24, no. 2, pp. 125–136, 1998, doi: 10.1109/32.666826.

[21] J. Arlat *et al.*, “Fault Injection for Dependability Validation,” *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 166–182, Feb. 1990, doi: 10.1109/32.44380.

[22] R. K. Iyer, “Experimental Evaluation,” *Proc. 25th International Symposium on Fault-Tolerant Computing, FTCS-25, Pasadena, California*, no. special issue, pp. 115–132, 1995.

[23] U. Gunneflo, J. Karlsson, and J. Torin, “Evaluation of error detection schemes using fault injection by heavy-ion radiation,” *1989 The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pp. 340–347, Jun. 1989.

[24] H. Madeira, M. Rela, F. Moreira, and J. Silva, “RIFLE: A General Purpose Pin-Level Fault Injector,” in *Proc. First European Dependable Computing Conference*, Oct. 1994, pp. 199–216.

[25] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, and J. Reisinger, “Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture,” in *Proceedings of the Fifth IFIP Working Conf. Dependable Computing for Critical Applications, DCCA-5*, 1995, pp. 150–151.

[26] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, “FERRARI: A tool for the validation of system dependability properties,” *FTCS 1992 - 22nd Annual International Symposium on Fault-Tolerant Computing*, pp. 336–344, 1992, doi: 10.1109/FTCS.1992.243567.

[27] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, “GOOFI: Generic object-oriented fault injection tool,” *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 83–88, 2001, doi: 10.1109/DSN.2001.941394.

[28] J. Christmansson and R. Chillarege, “Generation of an error set that emulates software faults based on field data,” *Proceedings - Annual International Conference on Fault-Tolerant Computing*, pp. 304–313, 1996, doi: 10.1109/FTCS.1996.534615.

[29] J. A. Durães and H. S. Madeira, “Emulation of software faults: A field data study and a practical approach,” *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 849–867, Nov. 2006, doi: 10.1109/TSE.2006.113.

[30] R. Natella, D. Cotroneo, and H. S. Madeira, “Assessing Dependability with Software Fault Injection,” *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, Feb. 2016, doi: 10.1145/2841425.

[31] N. Silva, M. Vieira, and D. Ricci, “Consolidated View on Space Software Engineering Problems-An Empirical Study,” *DASIA 2015-DAta Systems in Aerospace*, 2015.

[32] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, “An Empirical Study of the Impact of Single and Multiple Bit-Flip Errors in Programs,” *IEEE Transactions on Dependable and Secure Computing*, 2020, doi: 10.1109/TDSC.2020.3043023.

[33] A. Pereira, H. Madeira, and A. de Paula, “Experimental Validation of the Error Detection Mechanisms of the On-board Computer of the SSR Brazilian Satellite,” *(in Portuguese) VII Symposium on Fault Tolerant Computing, SCTF-7*, 1997.

[34] D. di Leo, F. Ayatolahi, B. Sangchoolie, J. Karlsson, and R. Johansson, “On the Impact of Hardware Faults – An Investigation of the Relationship between Workload Inputs and Failure Mode Distributions,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture*

Notes in Bioinformatics), vol. 7612 LNCS, pp. 198–209, 2012, doi: 10.1007/978-3-642-33678-2_17.

[35] M. Portela-García, C. López-Ongil, M. García-Valderas, and L. Entrena, “A rapid fault injection approach for measuring SEU sensitivity in complex processors,” *Proceedings - IOLTS 2007 13th IEEE International On-Line Testing Symposium*, pp. 101–106, 2007, doi: 10.1109/IOLTS.2007.9.

[36] M. Portela-García, C. López-Ongil, M. García Valderas, and L. Entrena, “Fault injection in modern microprocessors using on-chip debugging infrastructures,” *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 2, pp. 308–314, 2011, doi: 10.1109/TDSC.2010.50.

[37] M. Ebrahimi, A. Mohammadi, A. Ejlali, and S. G. Miremadi, “A fast, flexible, and easy-to-develop FPGA-based fault injection technique,” *Microelectronics Reliability*, vol. 54, no. 5, pp. 1000–1008, May 2014, doi: 10.1016/J.MICROREL.2014.01.002.

[38] H. Ziade, R. Ayoubi, and R. Velazco, “A Survey on Fault Injection Techniques,” *The International Arab Journal of Information Technology*, vol. 1, pp. 171–186, Jul. 2004.

[39] R. M. Tawfeek, M. G. Egila, Y. Alkabani, and I. M. Hafez, “Fault injection for FPGA applications in the space,” *Proceedings of ICCES 2017 12th International Conference on Computer Engineering and Systems*, vol. 2018-January, pp. 390–395, Jan. 2018, doi: 10.1109/ICCES.2017.8275338.

[40] J. L. Nunes, T. Pecserke, J. C. Cunha, and M. Zenha-Rela, “FIRED - Fault Injector for Reconfigurable Embedded Devices,” *Proceedings - 2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing, PRDC 2015*, pp. 1–10, Jan. 2016, doi: 10.1109/PRDC.2015.43.

[41] X. Meng, Z. Shao, J. Xu, Q. Tan, N. Zhang, and H. Zhang, “SEInjector: A dynamic fault injection tool for soft errors on x86,” *2017 International Conference on Computer Systems, Electronics and Control, ICCSEC 2017*, pp. 1492–1495, Aug. 2018, doi: 10.1109/ICCSEC.2017.8446693.

[42] S. Winter, T. Piper, O. Schwahn, R. Natella, N. Suri, and D. Cotroneo, “GRINDER: On Reusability of Fault Injection Tools,” *Proceedings - 10th International Workshop on Automation of Software Test, AST 2015*, pp. 75–79, Jul. 2015, doi: 10.1109/AST.2015.22.

[43] L. L. Pullum, *Software Fault Tolerance Techniques and Implementation*. USA: Artech House, Inc., 2001.

[44] M. Yang, G. Hua, Y. Feng, and J. Gong, *Fault-Tolerance Techniques for Spacecraft Control Computers*. 2017. doi: 10.1002/9781119107392.

[45] S. Mukherjee, *Architecture Design for Soft Errors*. 2008. doi: 10.1016/B978-0-12-369529-1.X5001-0.

[46] N. Murphy, “Watchdog Timers,” in *Embedded Systems Programming*, 2000, p. 112.

[47] M. S. Hefny and H. H. Amer, “Design of an improved watchdog circuit for microcontroller-based systems,” *Proceedings of the International Conference on Microelectronics, ICM*, vol. 2000-January, pp. 165–168, 1999, doi: 10.1109/ICM.2000.884831.

[48] V. B. Prasad, “Fault tolerant digital systems,” *IEEE Potentials*, vol. 8, no. 1, pp. 17–21, 1989, doi: 10.1109/45.31576.

[49] C. H. Stapper, V. K. Jain, and V. K. Jain, *Defect and Fault Tolerance in VLSI Systems*, 1st ed., vol. 2. New York, NY: Springer, 1990. doi: https://doi.org/10.1007/978-1-4757-9957-6.

[50] Liming Chen and A. Avizienis, “N-VERSION PROGRAMMINC: A FAULT-TOLERANCE APPROACH TO RELlABlLlTY OF SOFTWARE OPERATlON,” Aug. 1995, doi: 10.1109/FTCSH.1995.532621.

[51] V. Bharathi, “N-Version programming method of Software Fault Tolerance: A Critical Review,” *National Conference on Nonlinear Systems & Dynamics*, pp. 173–176, 2003.

[52] J. C. Knight and N. G. Leveson, “An experimental evaluation of the assumption of independence in multiversion programming,” *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, pp. 96–109, Jan. 1986, doi: 10.1109/TSE.1986.6312924.

[53] J. C. Knight and N. G. Leveson, “A reply to the criticisms of the Knight & Leveson experiment,” *ACM SIGSOFT Software Engineering Notes*, vol. 15, no. 1, pp. 24–35, Jan. 1990, doi: 10.1145/382294.382710.

[54] B. Randell, “System Structure for Software Fault Tolerance,” *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 220–232, 1975, doi: 10.1109/TSE.1975.6312842.

[55] IEEE, “IEEE Standard for Test Access Port and Boundary-Scan Architecture,” *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)*, pp. 1–444, May 2013, doi: 10.1109/IEEESTD.2013.6515989.

[56] OpenOCD, "Open On-Chip Debugger: OpenOCD User's Guide," 2022. https://openocd.org/doc/pdf/openocd.pdf (accessed Jul. 04, 2022).

[57] D. L. Domenicoand, F. Ayatolahi, B. Sangchoolie, J. Karlsson, and R. Johansson, "On the Impact of Hardware Faults – An Investigation of the Relationship between Workload Inputs and Failure Mode Distributions," in *Computer Safety, Reliability, and Security*, 2012, pp. 198–209.

[58] H. P. Zima, M. L. James, and P. L. Springer, "Fault-tolerant on-board computing for robotic space missions," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 17, pp. 2192–2204, Dec. 2011, doi: 10.1002/CPE.1768.

[59] D. Briere and P. Traverse, "Airbus A320/A330/340 electrical flight controls a family of fault-tolerant systems," *Digest of Papers - International Symposium on Fault-Tolerant Computing*, pp. 616–623, 1993, doi: 10.1109/FTCS.1993.627364.

[60] Y. C. Yeh, "Safety critical avionics for the 777 primary flight controls system," *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, vol. 1, 2001, doi: 10.1109/DASC.2001.963311.

[61] T. C. Bressoud, "TFT: A software system for application-transparent fault tolerance," *Digest of Papers - 28th Annual International Symposium on Fault-Tolerant Computing, FTCS 1998*, vol. 1998-January, pp. 128–137, 1998, doi: 10.1109/FTCS.1998.689462.

[62] B. Hasircioglu, Y.-A. Pignolet, and T. Sivanthi, "Transparent Fault Tolerance for Real-Time Automation Systems," *Proceedings of the 1st International Workshop on Internet of People, Assistive Robots and Things*, pp. 7–12, 2018, doi: 10.1145/3215525.3215538.

[63] K. Wilken and J. P. Shen, "Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Control Errors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 6, pp. 629–641, 1990, doi: 10.1109/43.55193.

[64] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, "Random additive signature monitoring for control flow error detection," *IEEE Transactions on Reliability*, vol. 66, no. 4, pp. 1178–1192, Dec. 2017, doi: 10.1109/TR.2017.2754548.

[65] S. A. Jacklin, "Survey of Verification and Validation Techniques for Small Satellite Software Development," *Space Tech Expo Conference*, May 2015.

[66] B. W. Boehm, "Guidelines for verifying and validating software requirements and design specification," *In Proceedings of the European Conference on Applied*

Information Technology of the International Federation for Information Processing (Euro IFIP), vol. 1, pp. 711–719, 1979.

[67] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*, 2nd ed. Elsevier, 2020.

[68] M. R. Lyu, *Software Fault Tolerance*, 1st Edition. John Wiley & Sons Ltd, 1995.

[69] A. C. R. Alves, S. M. Dias, K. I. P. de M. Queiroz, M. J. M. de Carvalho, and J. M. L. Duarte, “CONASAT-0: VISÃO GERAL DO NANOSSATÉLITE DESENVOLVIDO,” 2019. doi: 10.29327/2CAB2019.224823.

Appendices

Appendix A – CubeSatFI Functional Requirements

Define experiment information

Primary Actor: User | **Use Case ID:** 1.1

Scope: Fault injection campaign generation

Level: Sea

Stakeholders and Interests:

- INPE: The main interest stakeholder in the development of the tool, aiming to use it in the development of their CubeSats.
- CubeSat Practitioners: Can build more reliable CubeSat software applications.

Preconditions:

- The user must be in the Definition Campaign screen (after selecting the option in the navigation menu).
- The intended target system must be previously selected on the tool options page.

Minimal Guarantee:

- The system notifies the user if some field was not defined as expected.

Success Guarantee:

- The data is validated, and the campaign can be generated.

Main Success Scenario:

1. The system presents the Definition Campaign screen.
2. The user inserts the information related to the fault injection campaign: campaign name, short description, number of faults to generate, number of bitflips per fault, and the name of the person responsible for the campaign on proper fields.
3. The system presents the registers map of the target processor.
4. The user inserts select the target registers that want to affect by selecting the exact bit positions that want to affect (i.e., bit flipping).
5. The system presents the triggers definition options.
6. The user selects the time-based trigger option.
7. The system presents the fields for insertion of three different moments (start of the injection window, end of the injection window, and end of the injection run).

8. The user defines the three moments. 9. The system validates all the data.
Extensions: 2.a. In addition to the information described on point 2, the user also defines a seed (to be used on pseudo-random number generation). 6.a. The user selects the spatial-based trigger. - 6.a.1. The system presents the fields for insertion of a range of memory addresses and a moment (that represents the end of the injection run). - 6.a.2. The user defines the range and the moment. - 6.a.2. The use case continues on point 9 of the main success scenario.

Table 2 - UC 1.1: Define experiment information

Generate campaign	
Primary Actor: User	**Use Case ID:** 1.2
Scope: Fault injection campaign generation	
Level: Sea	
Stakeholders and Interests: • INPE: The main interest stakeholder in the development of the tool, aiming to use it in the development of their CubeSats. • CubeSat Practitioners: Can build more reliable CubeSat software applications.	
Preconditions: • The user must be on the Definition Campaign screen. • The intended target system must be previously selected on the tool options page. • All the mandatory fields must be filled with the respective information.	
Minimal Guarantee: • The system notifies the user if some error occurs during the generation of the campaign.	
Success Guarantee: • The campaign is generated, and all the data is saved for further execution.	
Main Success Scenario: 1. The user generates a faulty campaign. 2. The system presents a saving location window. 3. The user chooses the location to save the campaign. 4. The system saves the faults on a .csv file and the campaign configuration on a .json file. Informs the user with a success message.	
Extensions: 1.a. The user generates a Golden Run campaign. 2.a. The user aborts the campaign generation by moving to another screen. - 2.a.1. The system asks for confirmation of abort and informs the user that all the data will be lost. - 2.a.2. The user confirms that wants to abort the campaign generation. - 2.a.3. The system discards all the data and moves to the screen pretended. 2.b. The user aborts the campaign generation by moving to another screen.	

	- 2.a.1. The system asks for confirmation of abort and informs the user that all the data will be lost. - 2.a.2. The user wants to continue the campaign generation. - 2.a.3. The system maintains all the data and the user can continue the campaign generation.

Table 3 - UC 1.2: Generate campaign

Import fault injection campaign information	
Primary Actor: User	**Use Case ID:** 1.3
Scope: Fault injection campaign generation	
Level: Sea	
Stakeholders and Interests: • INPE: The main interest stakeholder in the development of the tool, aiming to use it in the development of their CubeSats. • CubeSat Practitioners: Can build more reliable CubeSat software applications.	
Preconditions: • The user must be in the Definition Campaign screen (after selecting the option in the navigation menu). • The intended target system must be previously selected on the tool options page.	
Minimal Guarantee: • The system notifies the user if some error occurs during the information importation.	
Success Guarantee: • The information is loaded, and the fields of the Definition Campaign screen are filled.	
Main Success Scenario: 1. The user decides to import an existent configuration. 2. ***Use Case 1.0.0*** is done at this point. 3. The system fields the corresponding fields with the information read from the configuration file.	

Table 4 - UC 1.3: Import fault injection campaign information

Search fault injection campaign file	
Primary Actor: User	**Use Case ID:** 1.0.0
Scope: Fault injection campaign generation	
Level: Sea	
Stakeholders and Interests: • INPE: The main interest stakeholder in the development of the tool, aiming to use it in the development of their CubeSats. • CubeSat Practitioners: Can build more reliable CubeSat software applications.	
Preconditions: • To choose a fault injection campaign file is mandatory the existence of one file.	
Minimal Guarantee: • The system notifies the user if some error occurs during the information importation (for example, wrong file format).	
Success Guarantee: • The information is loaded without errors.	
Main Success Scenario: 1. The user decides to search for an existent configuration. 2. The system presents a select location window. 3. The user chooses the file location of the campaign. 4. The system read the campaign configuration from a JSON file.	

Table 5 – UC 1.0.0: Search fault injection campaign file

Start fault injection campaign	
Primary Actor: User	**Use Case ID:** 2.1
Scope: Fault injection campaign execution	
Level: Sea	
Stakeholders and Interests: • INPE: The main interest stakeholder in the development of the tool, aiming to use it in the development of their CubeSats. • CubeSat Practitioners: Can build more reliable CubeSat software applications.	
Preconditions: • A fault injection campaign must be defined and generated before. • The user must select a fault injection campaign from a list of campaigns or search for an existent one. • The correct target system must be selected before.	
Minimal Guarantee: • If an error occurs the user is notified and all the data until that moment is saved. Any data is lost.	
Success Guarantee: • The information is loaded without errors.	
Main Success Scenario: 1. The system presents the configurations of the fault injection campaign. 2. The user selects the USB interface to collect data. 3. The user starts the campaign. 4. The system initiates and establishes communication with the OpenOCD server and shows the execution screen. 5. The system prints information about the fault that is currently being injected and repeats this until the end of the campaign.	
Extensions: 2.a. The user refreshes the list of USB interfaces available. - 2.a.1. The system refreshes the list of USB interfaces available. - 2.a.2. The user selects the USB interface to collect data. 4.a. The system notifies the user that an error occurs during the initiation of the OpenOCD server.	

5.a. The system notifies the user that an error occurs during the injection of a fault. All data is saved, and the next fault is injected. 5.b. The system notifies the user that an error occurs with the communication with the OpenOCD server. All data is saved until that moment is saved, and the campaign is aborted.

Table 6 - UC 2.1: Start fault injection campaign

Pause fault injection campaign	
Primary Actor: User	**Use Case ID:** 2.2
Scope: Fault injection campaign execution	
Level: Sea	
Stakeholders and Interests: • INPE: The main interest stakeholder in the development of the tool, aiming to use it in the development of their CubeSats. • CubeSat Practitioners: Can build more reliable CubeSat software applications.	
Preconditions: • A fault injection campaign must be already running.	
Minimal Guarantee: • If an error occurs the user is notified and all the data until that moment is saved. Any data is lost.	
Success Guarantee: • The fault injection campaign is paused right after finishing the injection of the current fault.	
Main Success Scenario: 1. The user intends to pause the fault injection campaign. 2. The system waits until the end of the injection of the fault that is currently being injected. After that, pause the injection of faults and give feedback to the user.	
Extensions: 2.a. An error occurs and the system notifies the user. All data is saved until that moment is saved, and the campaign is aborted.	

Table 7 - UC 2.2: Pause fault injection campaign

Resume fault injection campaign	
Primary Actor: User	**Use Case ID:** 2.3
Scope: Fault injection campaign execution	
Level: Sea	
Stakeholders and Interests: • INPE: The main interest stakeholder in the development of the tool, aiming to use it in the development of their CubeSats. • CubeSat Practitioners: Can build more reliable CubeSat software applications.	
Preconditions: • A fault injection campaign must be paused.	
Minimal Guarantee: • If an error occurs the user is notified and all the data until that moment is saved. Any data is lost.	
Success Guarantee: • The fault injection campaign is resumed, and the next fault started to be injected.	
Main Success Scenario: 1. The user intends to resume the fault injection campaign. 2. The system starts the injection of faults again.	
Extensions: 2.a. An error occurs and the system notifies the user. All data is saved until that moment is saved, and the campaign is aborted.	

Table 8 - UC 2.3: Resume fault injection campaign

Abort fault injection campaign	
Primary Actor: User	**Use Case ID:** 2.4
Scope: Fault injection campaign execution	
Level: Sea	
Stakeholders and Interests: • INPE: The main interest stakeholder in the development of the tool, aiming to use it in the development of their CubeSats. • CubeSat Practitioners: Can build more reliable CubeSat software applications.	
Preconditions: • A fault injection campaign must be paused.	
Minimal Guarantee: • If an error occurs the user is notified and all the data until that moment is saved. Any data is lost.	
Success Guarantee: • The fault injection campaign is resumed, and the next fault started to be injected.	
Main Success Scenario: 1. The user intends to abort the fault injection campaign. 2. The system asks for confirmation. 3. The user confirms. 4. The system waits until the current fault is injected. After that, aborts the campaign and backs to the home page. All the data until that moment is saved.	
Extensions: 3.a. The user does not confirm the abortion. - 2.a.1. The system continues with the fault injection campaign.	

Table 9 - UC 2.4: Abort fault injection campaign

List fault injection campaigns	
Primary Actor: User	**Use Case ID:** 2.1.1
Scope: Fault injection campaign execution	
Level: Sea	
Stakeholders and Interests: • INPE: The main interest stakeholder in the development of the tool, aiming to use it in the development of their CubeSats. • CubeSat Practitioners: Can build more reliable CubeSat software applications.	
Preconditions: • The system must be on the home page.	
Minimal Guarantee: • If it does not exist any history of campaigns previously generated, the user is informed.	
Success Guarantee: • The system presents a list of the last fault injection campaigns generated.	
Main Success Scenario: 1. The system reads the last fault injection campaigns generated from a file. 2. The system put the list of faults on the screen.	
Extensions: 2.a. The system put a message on the screen, adverting that does not exist any campaigns history previously generated.	

Table 10 – UC 2.1.1: List fault injection campaigns

Edit fault injection campaign	
Primary Actor: User	**Use Case ID:** 3.0
Scope: Fault injection campaign generation	
Level: Sea	
Stakeholders and Interests: • INPE: The main interest stakeholder in the development of the tool, aiming to use it in the development of their CubeSats. • CubeSat Practitioners: Can build more reliable CubeSat software applications.	
Preconditions: • The system must present a list of fault injection campaigns previously generated.	
Minimal Guarantee: • If an error occurs the user is notified. • The campaign data is validated.	
Success Guarantee: • The user edits a campaign previously generated and, the details changed are validated by the system, and a new campaign can be generated.	
Main Success Scenario: 1. The user chooses one campaign to edit. 2. The system loads the data of the campaign previously selected and presents the data to the user. 3. The user changes one or more details of the campaign. 4. The system validates all the data.	

Table 11 - UC 3.0: Edit fault injection campaign

Choose target system	
Primary Actor: User	**Use Case ID:** 4.1
Scope: CubeSatFI Configuration	
Level: Sea	
Stakeholders and Interests: • INPE: The main interest stakeholder in the development of the tool, aiming to use it in the development of their CubeSats. • CubeSat Practitioners: Can build more reliable CubeSat software applications.	
Preconditions: • The system must be on the configuration screen. • The target systems (CubeSat board) available must be available in a configuration file.	
Minimal Guarantee: • The user only can select available target systems.	
Success Guarantee: • After the selection of the target system, the system will always assume that target system.	
Main Success Scenario: 1. The system reads the available targets from a configuration file and presents them. 2. The user selects one target system. 3. The system updates the current target system option on the configuration file.	

Table 12 - UC 4.1: Choose the target system

Choose CubeSatFI language	
Primary Actor: User	**Use Case ID:** 4.2
Scope: CubeSatFI Configuration	
Level: Sea	
Stakeholders and Interests: • INPE: The main interest stakeholder in the development of the tool, aiming to use it in the development of their CubeSats. • CubeSat Practitioners: Can build more reliable CubeSat software applications.	
Preconditions: • The system must be on the configuration screen. • The target systems (CubeSat board) available must be available in a configuration file.	
Minimal Guarantee: • The user only can select available target systems.	
Success Guarantee: • After the selection of the target system, the system will always assume that target system.	
Main Success Scenario: 1. The system reads the available targets from a configuration file and presents them. 2. The user selects one target system. 3. The system updates the current target system option on the configuration file.	

Table 13 - UC 4.2: Choose CubeSatFI language

Appendix B - Fault injection platform for affordable verification and validation of CubeSats software

Fault injection platform for affordable verification and validation of CubeSats software

David Paiva
CISUC
University of Coimbra
Coimbra, Portugal
davidpaiva.uc@gmail.com

José Marcelo Duarte
COENE
INPE
Natal, Brazil
jose.duarte@inpe.br

Raffael Lima
COENE
INPE
Natal, Brazil
raffael.sadite@inpe.br

Manoel Carvalho
COENE
INPE
Natal, Brazil
manoel.carvalho@inpe.br

Fátima Mattiello-Francisco
COEPE
INPE
São José dos Campos, Brasil
fatima.mattiello@inpe.br

Henrique Madeira
CISUC
University of Coimbra
Coimbra, Portugal
henrique@dei.uc.pt

***Abstract*—CubeSats and very small satellites represent an emergent trend in the space industry. These satellites use commercial off-the-shelf (COTS) components to reduce cost and take advantage of the performance/power consumption ratio of COTS, which is an order of magnitude better than the equivalent radiation hardened space grade components. Unfortunately, COTS components are susceptible to Single Event Upsets (SEU), which are transient errors caused by space radiation. This makes the study of the impact of faults caused by space radiation a mandatory step in the development of CubSats, in order to carefully evaluate weak points that must be strengthened through the use of specific software fault tolerance techniques. The fact that the impact of faults is strongly dependent on the software running on the COTS hardware indicates that the study of the impact of radiation faults must be carried out every time the CubeSat software has a major change, or even a minor update.**

This paper proposes CubeSatFI, a fault injection platform for CubeSats meant to facilitate the incorporation of this extra step in the Verification and Validation of CubeSats software. CubeSatFI allows the easy definition of fault injection campaigns that emulate the effects of space radiation. SEU are emulated realistically through bit-flip faults injected in the processor registers and in other locations of the CubeSat boards that can be reached by boundary-scan, which is available in CubeSat boards through JTAG Test Access Port. The execution of the fault injection campaigns is controlled by the CubeSatFI platform in a fully automated mode. The paper describes the architecture of the CubeSatFI platform, the fault models, and the general fault injection process. Additionally, the use of the CubeSatFI platform is demonstrated with a fault injection campaign for the EDC (Environment Data Collection), a payload system that will be used in a constellation of satellite from the Brazilian National Institute for Space Research (*Instituto Nacional de Pesquisas Espaciais* - INPE), providing a first realistic insight on the impact of faults in the EDC software.

Keywords*—*fault injection, soft errors, CubSats, COTS, verification and validation, software fault tolerance techniques

I. Introduction

The use of commercial off-the-shelf (COTS) components (both hardware and software) and COTS-based systems in mission-critical applications is an established trend in the computer industry. They offer a real opportunity to reduce development costs and deployment times, which greatly explains the growing interest in using COTS components in mission-critical systems. Additionally, COTS components normally benefit from a large installation base in a multitude of configurations, which is often considered as an effective "test in the field".

The use of COTS-based systems in space missions is particularly attractive, especially in the context of CubeSats and nanosatellites where very low cost and very quick development time are paramount goals. However, in spite of the advantages of COTS components (low cost, top performance, low energy consumption, readily available for purchase), the reality is that COTS are not usually designed for the stringent requirements of space missions. In fact, the use of hardware COTS components in space applications introduces new challenges, as COTS hardware components are susceptible to transient errors due to single event upsets (SEU) caused by space radiation. This means that the actual use of COTS components in space missions must be preceded by careful study of the impact of faults caused by space radiation on system behavior. This is a necessary step, even for Low Earth Orbit (LEO) (and low risk) CubeSat missions.

Hardware COTS components used in boards of CubeSats are sensitive to space radiation. That is a known fact, well documented in the semiconductors data sheets and abundantly evaluated by researchers and practitioners, using ground radiation and methods described in the standard ISO 21980:2020 [1], and even in on-orbit measurements (e.g., [2, 3]). Space radiation can cause a variety of effects in COTS microelectronics, ranging from permanent failures to transient faults, depending on the different sources of space radiation

| /$31.00 ©10.1109/LADC53747.2021.9672584

978-1-6654-7831-1/21/$31.00 ©2021 IEEE

and on the radiation exposure [4]. However, transient faults caused by single event upsets (SEU) are recognized as the major cause of component malfunctions in space [5], especially in the LEO used by CubeSats. In other words, the major risk resulting from space radiation in CubeSats is the increased rate of hardware transient faults that may cause erroneous behavior in the software running on CubeSat boards.

Obviously, CubeSats boards can be designed to reduce the probability of hardware transient faults due to SEU. For example, using better COTS components (often called COTS+ [6]) and/or including some hardware fault tolerance mechanisms in the design of the boards. But the use of full-fledged hardware fault tolerance (e.g., TMR [7]) would be prohibitive in terms of power consumption and weight, and in practice CubeSats boards only have lightweight mechanisms such as memory error detection and correction. CubeSat proposals with strong fault tolerance mechanisms are rare, but even the few ones available, such as a recent proposal presented in [8], rely on software fault tolerance techniques [9] (with some support from the COTS hardware architecture in case of [8]).

Two significant points emerge from the above discussion:

a) The evaluation of the impact of space radiation in CubeSats should be focused on the software side, as SEU induced hardware transient faults will be visible (and will have impact) at software level.

b) The use of software fault tolerance techniques seems the most promising approach to increase CubeSats resilience in terms of space radiation, while keeping the affordable budget, low energy, low mass, and easy to purchase hardware components of CubeSats.

This paper proposes a fault injection platform (CubeSatFI) designed to facilitate both **a)** the evaluation of the impact of space radiation in CubeSats software and **b)** the development and validation of software fault tolerance mechanisms to improve CubeSats resilience to SEU induced faults. The goal is to allow CubeSat developers to define fault injection campaigns that emulate the effects of space radiation, providing developers' teams with an effective tool to verify and validate CubeSat software in terms of SEU induced faults.

It is worth mentioning that a consistent result in fault injection studies is that the impact of transient faults on the software behavior (i.e., the failure modes observed) is highly dependent on the actual software under evaluation. In [10], for example, faults injected while the target system was running quite diverse software, selected from many application areas such as automotive, telecom, office, etc., shows differences in the percentages of failure modes observed higher than 70%, depending on the actual software running. In another work, where injected faults intended to emulate SEU induced faults in a NASA COTS based payload for scientific data processing onboard of the satellite, the differences observed on the fault impact for the different programs running on the system reached 45% for some failure modes [11].

The fact that the impact of transient faults is highly dependent on the actual software suggests that fault injection should be a mandatory step in the development of CubeSats software and, more specifically, an integral part of the software verification and validation process for CubeSats. Even after relatively minor updates in the CubeSat software that may change the software behavior in the presence of SEU induced faults, it is recommended to perform fault injection tests to (re)validate CubeSat resilience to space radiation. The proposed CubeSatFI platform is intended to facilitate the integration of fault injection as a mandatory step in the verification and validation of CubeSats software.

The paper is organized as follows: the next section presents some background on small satellites and related work on the use of fault injection for software verification and validation. Section III describes the CubeSatFI architecture and Section IV presents a use case of CubeSatFI and discusses the results. Section V concludes the paper.

II. BACKGROUND AND RELATED WORK

A. *Small satellites and CubeSats*

There is a wide variety of small satellite types, which has been almost exclusively used in low Earth orbits for applications such as remote sensing or communications. The most common type among small satellites is known as nanosatellite, which includes satellites with a mass of up to 10 kg. The number of launches of nanosatellites has grown significantly since 2012, as shown in Fig. 1. This is largely due to the popularization of CubeSat satellites following the CubeSat Design Specification (CDS), which is a standard (de facto) for mechanical design and interfacing for satellites [12]. This standard defined the 1U format, a 10cm cube edge for the satellites, and other formats derived from it, 1.5U, 2U, 3U, 6U, etc. This standardization effort has significantly reduced the costs of satellite development and launching. Satellite subsystems such as solar panels, antennas, on-board computer, power system, communication system and others started being sold as COTS, and the standardization of the interface between the satellite and the launcher also simplified the provision of the launching service.

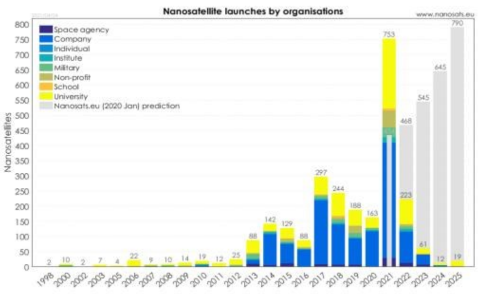


Fig. 1. Number of nanosatellite launches per year (from https://www.nanosats.eu/).

Appendix B - Fault injection platform for affordable verification and validation of CubeSats software

CubeSats are generally launched into low Earth orbits, at an altitude of up to 600 km. This is mainly due to its small size that limits the energy available in the power supply and makes it impossible to use high gain antennas. Launching CubeSats into high altitude orbits, such as the geostationary orbit around 36,000 km, would need high power transmitters to achieve high transmission rates, which are difficult to accommodate in the CDS standard. Among the popular CubeSat applications stand out communication services such as IoT and Space Internet, remote sensing with acquisition of images of the Earth and space and geolocation.

The use of COTS components in CubeSats is attractive because they largely outperform (in performance, cost, weight, etc.) components that are qualified for space applications. The spatial qualification process slows down components, as the space industry mainly limits the operating frequency and dynamic use of the processor memory cache to reduce the inherent risks of radiation suffered in space. COTS components also have the advantage of cost and ease of purchase, eliminating potential embargo issues due to the protected nature of many space applications.

B. *Fault injection for space applications*

Fault injection consists of "the deliberate insertion of artificial faults in a computer system or component in order to assess its behavior in the presence of faults and allow the characterization of specific dependability measures" of the target computer system [13]. This is a mature technology that has been established as an attractive way of validating specific fault handling mechanisms and as an effective technique to provide experimental data for the estimation of fault-tolerant system measures, such as fault coverage and error detection latency [14, 15].

There is a wide variety of techniques to inject faults in computer systems. First proposals of fault injection techniques used heavy-ion radiation into processor chips [16], pin-level fault injection [17], or electromagnetic interference [18]. However, the increased complexity of computer systems made these techniques nearly impossible to apply, not only because of the inherent difficulties of controlling the injection process in highly complex processors but also because of the difficulties in the collection of high-quality information on the target system behavior after the injection of the faults. In practice, the initial fault injection techniques have been replaced by the emulation of faults through software mechanisms, which is known as SWIFI (Software Implemented Fault Injection).

One of the first SWIFI proposals was [19], which was subsequently replaced by more effective and less intrusive methods such as [13, 20] that became the standard de facto in fault injection. The key idea of SWIFI tools is to emulate hardware faults through software, using very small interruption response routines, with just a few instructions (or scan-chain debugging resources in [20]) that insert bit-flip errors in very low-level structures (e.g., processor registers, memory, or busses, among others).

Initially, fault injection techniques only emulated (realistically) hardware faults through bit-flip and stuck-at bit fault models. One of the first studies that investigated the possibility of emulation of software faults (i.e., software bugs) using fault injection tools was published in [21]. The first practical approach to inject realistic software faults (that emulates real bugs found in deployed software) was presented in [22]. A relatively recent survey on software fault injection techniques can be found in [23].

Concerning the injection of faults that emulate the effects of SEU, the heavy-ion methods are in fact a direct injection of real radiation induced faults [16]. But this approach was replaced by SWIFI methods [13] since it cannot be used in modern processors. SWIFI techniques have been widely used by space agencies such as ABS [24], NASA [11] or ESA [25].

The CubeSatFI fault injection platform proposed in this paper uses SWIFI techniques inspired on [13] and, particularly, on scan chain approaches proposed in [20, 26].

III. CUBESATFI EXPERIMENTAL SETUP AND INJECTION APPROACH

The CubeSatFI fault injection platform was designed to evaluate the behavior of software running on top of COTS boards of nanosatellites and CubeSats in the presence of space radiation. The use of COTS components in the manufacturing of satellites, makes them vulnerable to the occurrence of a Single Event Upset (SEU) that can compromise the correct functioning of the satellites [5]. With that in mind, it is important to emulate these events and evaluate its consequences, in order to improve CubeSats software and build reliable satellite systems that can tolerate the effects of space radiation.

A. *Experimental setup and overview of the fault injection approach*

Fig. 2 shows the CubeSatFI fault injector setup. The Host PC runs the injection tool that uses the Open On-Chip Debugger (OpenOCD) [27] to communicate with the target system and perform the injection of the faults using the IEEE 1149.1 standard for boundary-scan [28] available in CubeSat boards (our target systems). The CubeSatFI running on the host PC uses the debugging and boundary-scan features available through the Joint Test Action Group (JTAG) adapter [28] to get access to the processor registers and other internal structures via the JTAG Test Access Port (TAP) of the target system.

The basic idea to perform fault injection using the IEEE 1149.1 standard for boundary-scan consists of using JTAG commands through OpenOCD to interrupt the normal execution of the program running on the target system and to get a copy of the internal state of the processor and other internal data included in the scan chain of the target system. Then, one or more bits of such internal state can be corrupted (according to the fault models described in subsection B) and the internal state is put back again, including the error caused by the emulated fault. After that, the normal execution of the software is resumed and the impact of the emulated fault in the target system is evaluated. The fault injection algorithm is discussed further on in more detail, in subsection B. This approach has already been used successfully to inject faults [20] and has the advantage of using the features available in

Appendix B - Fault injection platform for affordable verification and validation of CubeSats software the target system for testing purposes to inject faults with minimal perturbation of the system (other than the injected fault). Since most of the CubeSat boards include JTAG and TAP port, the proposed CubeSatFI can be used in most of the CubeSat satellites.

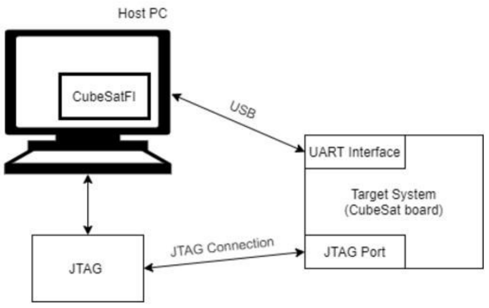


Fig. 2. CubeSatFI fault injection setup

The CubeSatFI functionalities include two big groups: **a)** fault injection campaign generation and **b)** fault injection campaign execution. The fault injection campaign generation allows the user to define controlled experiments through the specification of the number of faults to inject, type of faults to be injected, fault trigger conditions, among other things. The data describing each fault injection campaign is stored in a file. The fault injection campaign execution controls automatically the fault injection process (i.e., no user intervention is needed) and executes all the steps required to inject each fault and collect the relevant data, according to the fault injection workflow (see details in the next subsection). During the campaign execution, the OpenOCD server is launched in the Host PC and it is responsible for receiving all the instructions from the CubeSatFI, forwarding them to the target system through JTAG, and, consequently, receiving the respective responses.

The target behavior after the injection of each fault is collected and saved in a file. The information collected depends on the actual scenario in which the target system is being used and the purpose of the fault injection campaign. Since the results obtained by fault injection depend on the software running on top of COTS, the collection of results should be defined considering the specific functionalities of the system under testing and the testing objectives. The file with the fault injection results stored at the end of each campaign is analyzed using external statistical tools such as Excel or R.

B. *Fault injection model and injection workflow*

Faults are described by two groups of parameters, following well established practice in the fault injection area:

- **Fault type**: indicates the exact location of the fault in the target system and the number of bits affected (single bit-flip or multiple bit-flips). Only bit-flip faults are considered, since this is a well-established fault model for hardware faults induced by SEU [4, 5].
- **Fault trigger**: indicates the exact moment/conditions when the fault should be injected.

In the current version of the CubeSatFI platform, faults are injected in any bit of any register of the target processor and consists of single bit-flip or multiple bit-flips that emulate SEU and radiation bursts. The current fault triggers only consider the time domain, particularly the injection of faults randomly in the injection window (see Fig. 3). This is the basic method to get the statistical effects of faults induced by space radiation, as the injected faults are distributed at random in both the space (i.e., registers and register bits) and time domains. The next version of CubeSatFI will include faults in other processor areas (using the emulation of faults in internal units such as the ALU, internal bus, instruction decoding, etc., through a technique proposed in [13]) and in other elements of the target system accessible through boundary-scan, and more elaborated fault trigger modes such as injection of the faults when the program executes a given instruction or reads/writes a given memory address. Also, "inject on read" and "inject on write" [10, 20] fault triggers are also planned for the next CubeSatFI version.

Fig. 3 presents the principal steps of the injection workflow. Resetting the target is the first step of each injection run (we call "injection run" to the sequence of steps needed to inject a fault and collect results on the impact of such fault) to assure that the results in each fault are not "contaminated" by the effects of the previous fault. Once resetting the target system, and until the end of each injection run, there are some important moments depicted in Fig. 3 that require some explanations:

- **Start of the Injection Window (T0)**: From this moment the fault can be injected. This time indicates the start of a run injection window.
- **Injection (Tinj)**: When the fault is actually injected in the target system. In time-based fault triggers, this moment is calculated randomly and corresponds to a time between the T0 and T1.
- **End of the Injection Window (T1)**: The fault can be injected until this moment. This time indicates the end of the run injection window.
- **End of the Injection Run (Tend)**: Indicates the end of the injection run. The time between T1 and Tend (Tend -T1) is exclusive to save information about the target system's behavior after it has worked for some time with the fault.

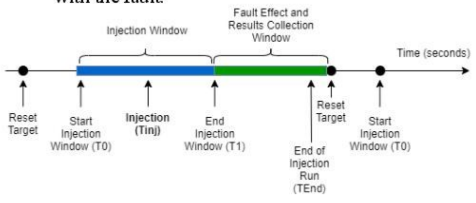

Injection Window
Fault Effect and Results Collection Window
Time (seconds)
Reset Target
Start Injection Window (T0)
Injection (Tinj)
End Injection Window (T1)
End of Injection Run (TEnd)
Reset Target
Start Injection Window (T0)

Fig. 3. Fault Injection Workflow

The collection of the results is the last step of one injection run and is done after some time interval to assure that the system works for some time after the injection of the fault, to be possible to measure the impact of the fault in the target system. In the case of the scenario tested (see section IV), for example, the results collected include the messages decoded by the target system (messages sent by a ground station) to allow the evaluation of failure modes, as well as information stored in the Program Counter register in the moment when the fault was injected, to allow knowing the exact program area directly affected by the fault.

```
public class ExecutionTool {

  private TelnetClient telnetClient;
  private List<Fault> faults;

  // ...

  private void loopInjectionCampaign(){
    for (Fault fault : this.faults) {
      // Start injection Run
      injectFault(fault);
    }
  }

  private void injectFault(Fault fault){
    String targetRegister = getRegisterID(fault.getRegister());
    // Reset the target
    telnetClient.resetProcessor();
    // Wait some time until the injection moment
    waitSomeTime(fault.getTinj());
    // Halt Target
    telnetClient.haltProcessor();
    // Get Register Value
    String binaryRegValue = telnetClient.getRegisterValue(targetRegister);
    // Change the register value
    String hexRegValue = this.xorBitFliping(binaryRegValue, fault.getMask());
    // Insert the fault into the target
    telnetClient.modifyRegisterValue(targetRegister, hexRegValue);
    //Get Program Counter register Value
    String programCounter = telnetClient.getProgramCounter();
    // Resume the Target Execution
    telnetClient.resumeProcessor();
    // Wait some time until the end of the injection Run
    waitSomeTime(fault.getT1()-fault.getTinj());
  }

  // ...
}
```

Fig. 4. Fault Injection Algorithm

Fig. 4 shows the algorithm for the injection of a campaign. The method loopInjectionCampaign goes across the list of faults defined in the campaign and injects the faults one by one. The method injectFault receives a fault and starts by restarting the target and waiting until the injection time (Tinj). When this moment is reached, it sends the command to halt the target system. After this moment, the value of the register to be affected by the fault is obtained, and the mask with the fault is applied to emulate the fault through the insertion of an error with the minimal perturbation possible. Once the error is injected, the command that resumes the target operation is sent and the program continues the execution in the target system. At this stage, the error caused by the injected fault may propagate and cause impact on the target system behavior. This execution under faulty conditions continues until the end of the injection run.

C. *User interface and features*

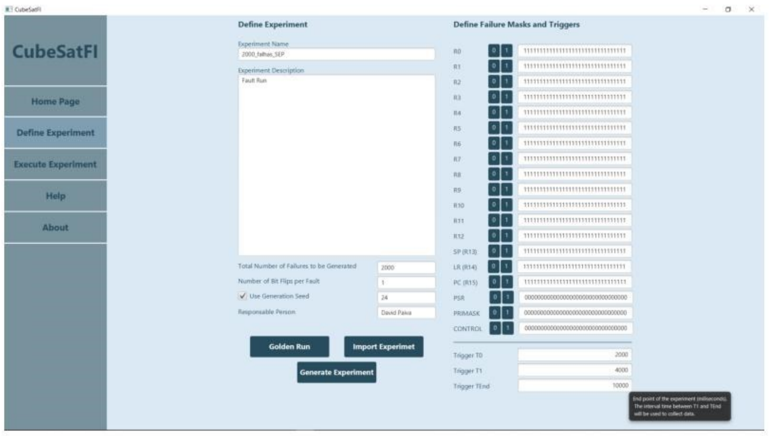
Fig. 5 shows the campaign definition screen where the fault injection campaigns are defined. In the current version of CubeSatFI, the user can define fault injection campaigns that emulate SEU affecting processor registers of the processor of the target CubeSat board. On the central area of the form shown in Fig. 2, the user defines the experiment name, description, number of faults to be injected in the campaign, number of bitflips per fault, the seed to use in the random number generation (this is important to decide whether the experiment is reproducible or not), and the name of the person responsible for the fault injection campaign. On the right-hand side of the form, the user defines the masks to apply to the target processor registers that define the register and/or registers that can be selected to inject faults and the bits inside each register that can be affected by the faults. In the bottom on the right hand side of the form, the user selects the fault triggers (in time domain) and defines the timing for the different moments of the fault injection workflow, as defined in Fig, 3. After providing all this data, the user can click on the button "Generate Experiment" to ask CubeSatFI to generate the description of the faults, which are stored in a file with all the information about the injection campaign.

The button "Import Experiment" allows the user to select a file describing a fault injection campaign previously defined and loads such file. After loading the file (or after defining a new fault injection campaign), the user can always change any of the parameters that define the fault injection campaign or can select the execution of a "Golden Run" in the bottom of the central part of the screen. The golden run simply runs the workload one or more times to record the correct behavior of the target system. In practice, the golden run is similar to a fault injection run, with the capital difference that no fault is injected. The behavior of the target system recorded during the execution of the golden run will be used later on to evaluate the failure modes of the target system after each injection run.

After defining a fault injection campaign, the user can select "Execute Experiment" at the left-hand side menu in order to start a fault injection campaign. As already mentioned, this step is fully automatic, follows the workflow presented in Fig. 3, and is totally controlled by the CubeSatFI. The central window in Fig. 6 displays contextual information about the fault injection process, namely the number of the fault currently being injected. During the execution of the fault injection campaign, the user can simply pause the fault injection process by clicking on the button "Pause" (the system will suspend the fault injection process after completing the fault currently being injected). To resume the injection process, later on, the user should click on the "Start" button. The user can also abort the fault injection campaign by clicking on the button "Abort".

Concerning the usability of the tool, a menu with the main features of the CubeSatFI is always present on the left-hand side of the screen. The feature currently selected is highlighted in a lighter color, making it easy to identify the selected functionality. Furthermore, to improve the user's experience, tooltips are available in most fields and buttons. These messages are displayed whenever the user puts the mouse over them. See an example in Fig. 5 on the bottom right-hand corner of the form.

Since the CubeSatFI is being developed in the context of international projects (see Acknowledgements at the end of the paper), the tool is multi-language. Currently, the user can select Portuguese or English at any time, and other idioms can be easily added, programmatically.



Define Experiment

Experiment Name: 2000_falhas_SEP

Experiment Description: Fault Run

Total Number of Failures to be Generated: 2000

Number of Bit Flips per Fault: 1

✓ Use Generation Seed: 24

Responsable Person: David Paiva

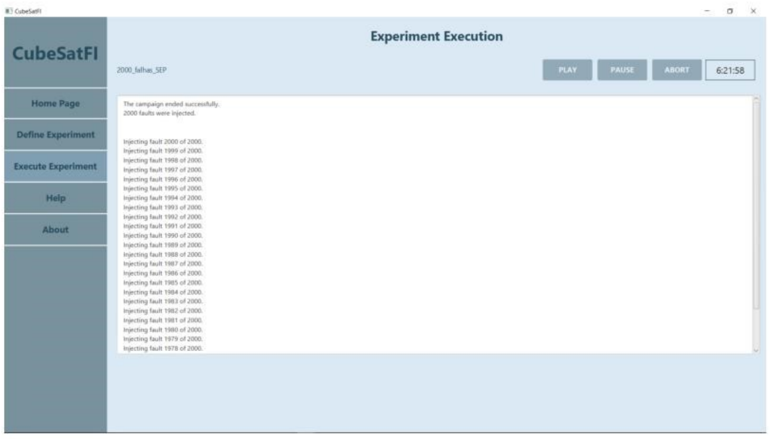
Define Failure Masks and Triggers

```
R0         11111111111111111111111111111111
R1         11111111111111111111111111111111
R2         11111111111111111111111111111111
R3         11111111111111111111111111111111
R4         11111111111111111111111111111111
R5         11111111111111111111111111111111
R6         11111111111111111111111111111111
R7         11111111111111111111111111111111
R8         11111111111111111111111111111111
R9         11111111111111111111111111111111
R10        11111111111111111111111111111111
R11        11111111111111111111111111111111
R12        11111111111111111111111111111111
SP (R13)   11111111111111111111111111111111
LR (R14)   11111111111111111111111111111111
PC (R15)   11111111111111111111111111111111
PSR        00000000000000000000000000000000
PRIMASK    00000000000000000000000000000000
CONTROL    00000000000000000000000000000000

Trigger T0                              2000
Trigger T1                              4000
Trigger TEnd                           10000
```

End point of the experiment (milliseconds). The interval time between T1 and TEnd will be used to collect data.

Fig. 5. Campaign Definition Screen



Experiment Execution

2000_falhas_SEP

6:21:58

```
The campaign ended successfully.
2000 faults were injected.


Injecting fault 2000 of 2000.
Injecting fault 1999 of 2000.
Injecting fault 1998 of 2000.
Injecting fault 1997 of 2000.
Injecting fault 1996 of 2000.
Injecting fault 1995 of 2000.
Injecting fault 1994 of 2000.
Injecting fault 1993 of 2000.
Injecting fault 1992 of 2000.
Injecting fault 1991 of 2000.
Injecting fault 1990 of 2000.
Injecting fault 1989 of 2000.
Injecting fault 1988 of 2000.
Injecting fault 1987 of 2000.
Injecting fault 1986 of 2000.
Injecting fault 1985 of 2000.
Injecting fault 1984 of 2000.
Injecting fault 1983 of 2000.
Injecting fault 1982 of 2000.
Injecting fault 1981 of 2000.
Injecting fault 1980 of 2000.
Injecting fault 1979 of 2000.
Injecting fault 1978 of 2000.
```

Fig. 6. Experiment Execution Screen

Finally, the CubeSatFI was designed to allow easy accommodation of additional features, as planned for the subsequent versions of the tool.

IV. HANDS-ON EXAMPLE OF CUBESATFI UTILIZATION

This section presents a use case of CubeSatFI. The results obtained provide some useful insights on the impact of the injected fault on the target system. For this very first example, we selected the Environmental Data Collector (EDC) [29], a CubeSat payload for the Brazilian Environmental Data Collection System. This payload is going to be used in all the nanosatellites from the CONASAT project [30].

A. *Experiment design*

The EDC is a multi-user RF receiver for a satellite message forwarding system. These systems offer sensor data transmission service in remote areas, such as environmental monitoring, wildlife tracking, vessel tracking, among others, with the lowest structural cost. These systems consist of ground platforms (GP), a satellite constellation, one or more receive stations (RS) and a data distribution center (DDC). The GPs transmit local sensor data through periodic messages, coded in RF burst transmissions. The satellite relays the received GP messages to a RS, when possible. Finally, the RSs transmits the GP messages to the DDC, which provides the cloud service for the system users. In this context, the EDC is the satellite payload that receives and decodes the GP messages. It does not have a transmitter. Therefore, the satellite on-board computer (OBC) must read the received message from the EDC and forward them to a RS multiplexing the data in one of its telemetry channels.

The EDC has a RF-Front-End unit that digitalizes the received RF signal and a processing unit that configures the RF-Front-End at system startup, decodes the received GP signals, and provides the interface with the OBC through a serial port. Besides the decoded messages, the EDC also provides housekeeping information to the OBC, such as supply current and voltage sensor measures, elapsed time since the last system reset, and others. The processing unit is implemented in an SoC FPGA (System on Chip - Field Programmable Gate Array), which has an internal hardwire microcontroller based on a Cortex-M3 processor. The signal decoding processing is split between the FPGA (hardware) and the processor (software), while the OBC interface and RF-Front-End configuration are fully implemented in software. The EDC software runs on top of the multi-task based FreeRTOS real-time operating system.

When the EDC is turned on, the OBC must initialize it by sending the Real Time Clock. A housekeeping frame must be requested for checking the temperature, electrical current, and electrical voltage sensors. This frame also indicates if the RF-Front-End configuration was successful. After this verification, signal decoding must be enabled, which starts disabled by default.

The OBC must periodically request the status of the decoded message buffer, in order to request the reading of the GP packages, if there are any messages. These packages have a variable length and are composed of a tag RTC time the message was received, a code that indicates an error in the decoding process, the frequency and amplitude of the received signal, the message length, the variable length GP message, and a checksum byte. At each reading of a GP package, the OBC must send a command to remove the package from the buffer, allowing the reading of another package. For telemetry, it is expected to send a housekeeping frame followed by a sequence of GP packages.

Fig. 7 shows a photo of the EDC experimental setup used to demonstrate the utilization of the proposed CubeSatFI fault injection platform. In addition to the elements already described for the target system (the EDC), the setup also includes the RF generator to emulate the communication with the satellite, a serial/USB convertor, the power supply and the host computer (the PC) that runs the CubeSatFI and the OpenOCD.

Since the goal of this experiment is to demonstrate the use of CubeSatFI to evaluate the impact of SEU induced faults in the EDC CubeSat board, we have defined a first fault injection campaign with 2000 faults injected at random, since the space radiation tends to affect the board in a random way. Faults are injected into any register of the processors, selected at random, and within the selected register for a given fault, the bit of the register affected by the fault was also selected at random. All the faults are single bit-flip faults, as this model is widely accepted as a realistic emulation of SEU faults. The trigger of each fault is also defined at random within the injection window (see Fig. 3). The injection window interval was defined as between 2 and 4 seconds. After the end of the injection window, the target system will be running to evaluate the effects of the fault for a period of 6 seconds. The messages decoded by the EDC are saved in a file (obviously, in some cases the injected fault causes the EDC to crash and the message decoding is interrupted).

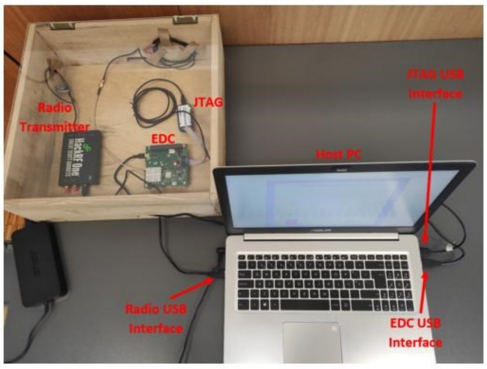


Fig. 7. Photo of the EDC Experimental Setup

B. Results and discussion

Fig. 8 shows the general impact of faults (i.e., failure modes) while EDC decodes messages from the point of view of the satellite on-board computer (OBC). The confidence intervals (shown in the numeric values in each bar of the chart) are calculated for 95% of confidence, using confidence intervals for proportions in binomial distributions (Bernoulli trials).

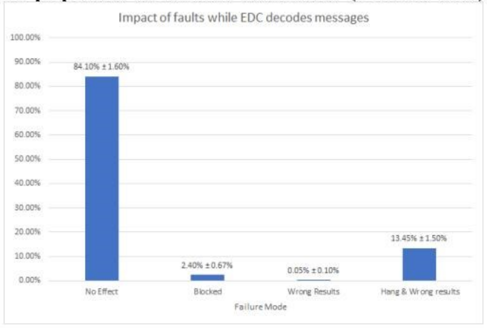


Fig. 8. Impact of faults while EDC decodes messages

The classification of failure modes was made based on the results obtained and include the following failure mode types:

- **No effect**: The fault had no visible impact on the system, which means that EDC continues to work normally, all the messages are well decoded and are sent correctly to the OBC.
- **Blocked**: The system blocks and stops sending decoded messages to the OBC. The only way to remove the target system from this failure mode is through a hard reset.

- **Wrong results**: The system sends messages with wrong information. The target system needs a hard reset after entering in this failure mode.
- **Hang & Wrong results**: After the injection the system hangs for a moment, and after a while starts o sending messages with wrong information. Needs a hard reset to leave this failure mode.

The results in Fig. 8 show that most of the faults (84.1%) had no effect on the behavior of the software running on the EDC. This result is not surprising if we consider the inherent redundancy existing in computer systems and in software. Furthermore, this high percentage of "no effect" faults have been consistently observed in many faults injection studies (e.g., [10, 13, 15, 26]), including in a fault injection study done with a COTS based payload system from a NASA project [11]. The more detailed analysis presented further on (about Fig. 9 results) explains (at least in part) this very high percentage of "No impact".

The analysis of the other failure modes shown in Fig. 8 shows also some interesting results. A very small percentage of faults caused wrong results (0.05%). Since the messages have a well-defined format, these wrong results are easy to detect. In other words, we have not observed failure modes that represent silent data corruption (SDC), which consists of erroneous results that are often not possible to detect and represent a serious risk.

The percentage of faults that crash the EDC software ("Blocked" failure mode) is also quite small (2.4%). In previous fault injection experiments reported in the literature (e.g., in [11]), this type of failure mode appeared in a much higher percentage of faults. One possible reason to explain the low percentage of "Blocked" failure modes in our experiment could be the fact that the EDC runs a very simple real-time operating system (FreeRTOS), since crashes are often caused by operating system crashes.

Still in the results shown in Fig. 8, 13.45% of the faults, the EDC shows a strange behavior, starting by not responding at all and then, after some time, starts to send messages with wrong values. The full understanding of this behavior would need a deeper analysis to identify the software idiosyncrasy that originates this behavior (and maybe find a way to make the software more robust in these particular cases). Although we have not analyzed the results in detail to try to explain this failure mode, it is worth noting that CubeSatFI records the exact program location affected by the fault (the value of the Program Counter when the fault was injected), allowing the detail analysis of the fault effect at low levels of the object code.

Fig. 9. shows the failure modes observed for the faults injected in each processor register. Two observations are quite evident: **a)** faults injected in the registers Program Counter (PC), Stack Pointer (SP), Link Register (LR) and R7 have a strong impact on the EDC software, and **b)** faults injected in many general-purpose registers (e.g., R1, R2, R4, R5, R6, R8, R9, R10, R11, R12) have no impact at all.

The PC, SP, and LR are special registers of the processor, therefore it is expected that any fault injected into one of these registers will cause the system to perform an incoherent behavior or even completely block the system. This is not a surprising result.

However, that fact that faults in many general registers have no impact is much more interesting. It means that the software running on the EDC rarely uses those registers or does not use them at all. This could be explained considering the way the C compiler (the EDC software was developed in C language) translates the source code into object code, which tends to use some preferential registers. In Fig. 9 we can observe that among general purpose registers, it seems that the compiler mainly used register R7 (and also a bit R0 and R3), as faults injected in other general-purpose registers had no effect. Obviously, this is highly dependent on the actual source code and also on the compiler optimization switches. **This is also the reason why the susceptibility of CubeSat boards to SEU induced faults is highly dependent on the actual software running on the target system**: if the software running on the EDC was more complex or the compiler switches are different, the percentage of "No effect" could drop dramatically.

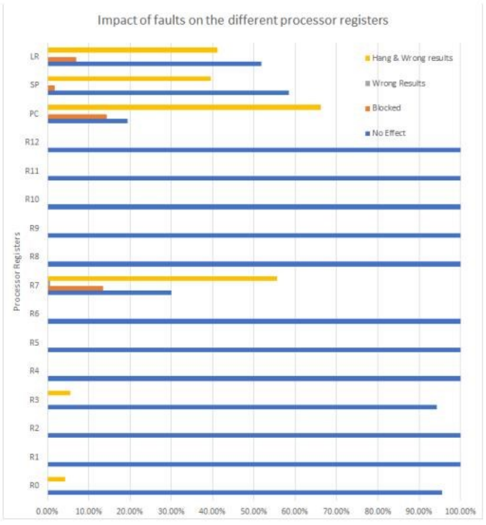


Fig. 9. Impact of faults on the different processor registers

Results shown in Fig. 9 provide "food for thought" and give some room for speculations/observations. **Clearly, the impact of SEU induced faults is very dependent on the actual software and Fig. 9 shows an important reason for that**, which is related to the way the processor resources (especially the registers) are used by the software. This suggests that fault injection campaigns should be executed as a routine procedure during the software development process, as part of the software verification and validation strategy. Even small changes in the software (or in the compiler switches) that lead to a different utilization of the processor registers by the object code could have a considerable impact on the software resilience in the presence of faults caused by space radiation.

Another observation from Fig. 9 (related to the fact that the compiler often uses just a few processors general registers) is that we can see the "free" registers as useful resources to develop software fault tolerant techniques. Given the nature of SEU (i.e., it is caused by a single particle that causes normally a one bit flip), if software fault tolerance techniques do not use the same registers used by the original software, the effectiveness of such software fault tolerance mechanisms could be much higher.

Fig. 10 shows the breakdown of the failure mode results according to different bits affected by the faults. A clear conclusion is that when the faults affect the 16 less significant bits groups ([1-8] and [9 -16]), the impact of faults is much higher than for the other bit positions. On average, when the faults are injected in the 16 less significant bits of the registers, the target system behaves outside the expected behavior (i.e., failure modes showing abnormal behavior) in about 20% of the faults. On the other hand, when the fault was injected in the first 8 most significant bits group ([17-24]), we observed that, on average, 87.55% of the faults have no impact on the target system. A similar result was observed for faults injected in the last 8 most significant bits group ([25-32]), as 91.68% of the faults had no impact on the target system behavior. In short, looking at the faults injected in the EDC system is possible to conclude that faults injected in the less significant bits lead to much more drastic wrong behavior in the system than faults injected in the second half of 16 most significant bits.

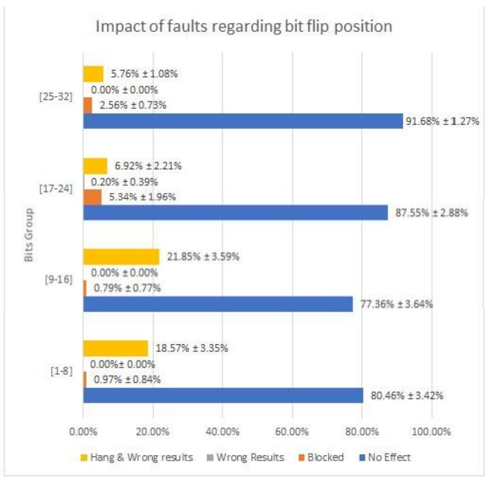


Fig. 10. Impact of faults regarding bit flip position

The fact that the CubeSatFI allows one to choose the precise bits and registers to be affected, opens the possibility to design more focused experiments and, consequently, evaluate in more detail specific erroneous behavior of the software running on the target system. Considering the data collected with the experiment presented before, it was decided to perform a second fault injection campaign of 1000 faults with random time fault triggers, in order to evaluate the EDC software behavior when the faults are injected on the less significant bits of the registers that show high fault impact. Fig. 11 shows the results of the faults injected on the less significant bits of the LR, SP, PC and R7 processor registers and shows that the less significant bits of these registers have a strong negative impact on the EDC behavior in the presence of faults. Faults injected in the less significant bits of the registers PC and R7, on average, cause more than 90% of wrong results, while the registers LR and SP show 68% and 75%, respectively.

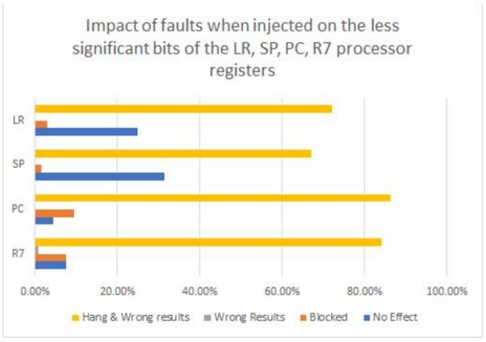


Fig. 11. Impact of faults when injected on the less significant bits of the LR, SP, PC, R7

The reported experiment showed that the CubeSatFI can effectively identify weak points of the target system concerning the impact of SEU caused by space radiation. These weak points are related to structural (i.e., hardware) aspects of the target systems, but the reported results also show that CubeSatFI has a great potential to help improving the resilience of the software running on CubeSats.

V. Conclusions and Future Work

This paper proposes CubeSatFI, a fault injection platform for CubeSat satellites. These satellites use commercial off-the-shelf (COTS) hardware components, which are susceptible to Single Event Upsets (SEU) caused by space radiation. The high rate of hardware transient faults in CubeSats represents an important risk. CubeSatFI allows the easy definition of fault injection campaigns that emulate SEU induced faults in CubeSat boards, providing effective means to carefully identify the impact of SEU faults and identify possible weak points in CubeSat software.

Although fault injection is a widely used technique in several industrial application areas, including in the space domain, the concrete application of fault injection in the CubeSat industry requires a new perspective and leads to new ways of using fault injection in the development of CubeSats and, more specifically, in the software verification and validation phases. A key idea is grounded on the fact that the impact of transient hardware faults in computer systems is highly dependent on the actual code running on such systems. When the code changes, the impact of faults could change drastically. In the case of CubeSats development, this means that the evaluation of the impact of SEU induced faults must be carried out every time the CubeSat software has a major change, or even a minor update. In other words, fault injection must be a mandatory step in the development of software for CubeSats.

A second key idea is that the negative impact of SEU induced faults in CubeSats boards should be mainly mitigated (or even tolerated) by software implemented fault tolerance techniques, given the strong restrictions in weight and power consumption of CubeSats that strongly limit other physical/hardware-based approaches. This represents a second and very important reason to use fault injection as a key element to evaluate the effectiveness of such software techniques designed to make CubeSat software more resilient to SEU. If those software techniques are meant to tolerate SEU faults, the most effective way to test them (and to evaluate their effectiveness) is simply injecting such types of faults.

The design of CubeSatFI as a practical tool to define fault injection experiments for CubeSats considers this dual utilization as part of the software verification and validation process and as a tool to evaluate the effectiveness of software implemented fault tolerance techniques.

The paper describes the architecture of CubeSatFI and presents a real use case of fault injection in the Environmental Data Collector (EDC), which is a CubeSat payload for the Brazilian Environmental Data Collection System. This payload is going to be used in all the nanosatellites from the CONASAT project. The presented use case demonstrates the use of CubeSatFI, but also shows that even a relatively simple fault injection campaign can provide valuable insights on the effect of SEU induced faults on CubeSat software. The results show the failure modes observed and demonstrate the type of fine grain analysis that can be done with the fault injection results obtained with CubeSatFI, showing the potential of the proposed fault injection platform.

CubeSatFI is being developed in the context of the European H2020 ADVANCE ("Addressing Verification and Validation Challenges in Future Cyber-Physical Systems") and VALU3S ("Verification and Validation of Automated Systems' Safety and Security") projects. The future versions of CubeSatFI will include additional fault types in other processor areas and in other elements of the target system accessible through boundary-scan, and more elaborated fault trigger modes such as injection of the faults when the program executes a given instruction or reads/writes a given memory address. Also, "inject on read" and "inject on write" fault triggers are also planned for the next CubeSatFI version.

Acknowledgment

This work was supported in part by the Grant CISUC-UID/CEC/00326/2020, funded in part by the European Social Fund, through the Regional Operational Program Centro 2020, and supported in part by the H2020, Marie-Curie project ADVANCE ("Addressing Verification and Validation Challenges in Future Cyber-Physical Systems"), project funded by the VALU3S ("Verification and Validation of Automated Systems' Safety and Security") project, supported by the European Commission, and in part by European Leadership (ECSEL) Joint Undertaking (JU) under Grant 876852, and in part by the JU from the European Union's Horizon 2020 Research and Innovation Programme.

References

[1] ISO 21980:2020, "Space systems — Evaluation of radiation effects on Commercial-Off-The-Shelf (COTS) parts for use on low-orbit satellites", ISO/TC 20/SC 14 Space systems and operations, first edition, January 2020.

[2] S. Shimhanda and T. Murase, "On-orbit Measurements of Radiation Effects on Commercial-Off-The-Shelf (COTS) Hardware for Small Satellites", Kyushu Institute of Technology, Kitakyushu, Japan, November 2019.

[3] R. García Alía *et al.*, "Simplified SEE Sensitivity Screening for COTS Components in Space," in *IEEE Transactions on Nuclear Science*, vol. 64, no. 2, pp. 882-890, Feb. 2017

[4] D. Sinclair and J. Dyer, "Radiation effects and COTS parts in SmallSats", 27th Annual AIAA/USU Conference on Small Satellites, SSC13-IV-3, 2013.

[5] R. Ecoffet, "Spacecraft Anomalies Associated with Radiation Effects", in RADECS 2013 Short Course Proceedings, Chap. VIII, 2013.

[6] T. Zednicek, "Commercial versus COTS+ versus Qualified Passive Components in Space Applications", Conference: ESA Space Passive Component DaysAt: ESA, ESTEC, Nordwijk, The Netherlands, Vol. 2nd SPCD, October 2016.

[7] A. Avizienis, J-C Laprie, B. Randell, C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing", IEEE Transactions on Dependable and Secure Computing, Vol. 1, No. 1, Jan.-March 2004.

[8] C. M. Fuchs et al., "Fault Tolerant Nanosatellite Computing on a Budget", 33rd Annual AIAA/USU Conference on Small Satellite, USA, 2019.

[9] W. Torres-Pomales, "Software Fault Tolerance: A Tutorial", NASA Langley Research Center, Hampton, Virginia, USA, NASA/TM-2000-210616, 2000.

[10] B. Sangchoolie, K. Pattabiraman and J. Karlsson, "An Empirical Study of the Impact of Single and Multiple Bit-Flip Errors in Programs," in IEEE Transactions on Dependable and Secure Computing, 2020.

[11] H. Madeira, R. Some, F. Moreira, D. Costa, D. Rennels, "Experimental evaluation of a COTS system for space applications", Int. Conference on Dependable Systems and Networks, DSN-2002, Bethesda, Maryland, USA, 2002.

[12] CubeSat Design Specification (1U – 12U), REV 14, CP-CDS-R14 https://static1.squarespace.com/static/5418c831e4b0fa4ecac1bacd/t/5f24997b6deea10cc52bb016/1596234122437/CDS+REV14+2020-07-31+DRAFT.pdf Accessed on August 8, 2021.

[13] J. Carreira, H. Madeira and J. G. Silva, "Xception: Software Fault Injection and Monitoring in Processor Functional Units", IEEE Transactions on Software Engineering, Vol. 24, No.2, Feb. 1998.

[14] J. Arlat et al., "Fault Injection for Dependability Validation: A Methodology and Some Applications," IEEE Transactions on Software Engineering, vol. 16, no. 2, pp. 166-182, Feb. 1990.

[15] R.K. Iyer, "Experimental Evaluation," Proc. 25th International Symposium on Fault-Tolerant Computing, FTCS-25, Pasadena, California, special issue, pp. 115-132, 1995.

[16] U. Gunneflo, J. Karlsson and J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation", *Proc. FTCS-19*, pp. 340-347, 1989-June.

[17] H. Madeira, M. Rela, F. Moreira, and J. Silva, "RIFLE: A General Purpose Pin-Level Fault Injector," Proc. First European Dependable Computing Conference, Berlin, pp. 199-216, October 1994.

[18] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, and J. Reisinger, "Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture," Proceedings of the Fifth IFIP Working Conf. Dependable Computing for Critical Applications, DCCA-5, Urbana-Champaign, Ill., pp. 150-151, 1995.

[19] G. Kanawati, N. Kanawati, and J. Abraham, "FERRARI: A Tool for the Validation of System Dependability Properties," FTCS-22, Digest of Papers, pp. 336-344, 1992

[20] J. Aidemark, J. Vinter, P. Folkesson and J. Karlsson, "GOOFI: generic object-oriented fault injection tool,", International Conference on Dependable Systems and Networks, DSN 2001, pp. 83-88, 2001.

[21] J. Christmansson and R. Chillarege, "Generation of an Error Set That Emulates Software Faults Based on Field Data," Proc. FTCS-27, Sendai, Japan, pp. 304-313, 1996.

[22] João A. Durães and Henrique S. Madeira "Emulation of Software Faults: A Field Data Study and a Practical Approach", IEEE Transactions on Software Engineering, vol. 32, no. 11, pp. 849-867, November 2006

[23] R. Natella, D. Cotroneo, and H. Madeira, "Assessing Dependability with Software Fault Injection: A Survey", ACM Computing Surveys, Volume 48 Issue 3, February 2016

[24] A. Pereira, H. Madeira, and A. de Paula, "Experimental Validation of the Error Detection Mechanisms of the On-board Computer of the SSR Brazilian Satellite", (in Portuguese) VII Symposium on Fault Tolerant Computing, SCTF-7, Paraiba, Brazil, 1997.

[25] N Silva, M Vieira, D Ricci, D Cotroneo, "Consolidated View on Space Software Engineering Problems-An Empirical Study", DASIA 2015-DAta Systems in Aerospace, 2015.

[26] D. Di Leo, F. Ayatolahi, B. Sangchoolie, J. Karlsson, R. Johansson, "On the Impact of Hardware Faults: An Investigation of the Relationship between Workload Inputs and Failure Mode Distributions", in Proceedings of the 31st International Conference on Computer Safety, Reliability, and Security (SAFECOMP), Magdeburg, Germany, 2012.

[27] OpenOCD - S. Oliver, O.Harboe, D. Ellis and D. Brownell, "Open On-Chip Debugger: OpenOCD User's Guide", August 2, 2021, available at http://openocd.org/doc/pdf/openocd.pdf, accessed on August 5, 2021.

[28] IEEE Std. 1149.1 - "Standard Test Access Port and Boundary-Scan Architecture", IEEE 1149.1 Working Group, March, 2012, https://grouper.ieee.org/groups/1149/1/, accessed on August 8, 2021.

[29] INPE. "Environmental Data Collector (EDC)". INPE, July 5, 2021. Accessed on: August 9, 2021. [Online]. Available: http://www.inpe.br/crn/projetos/edc.php

[30] K. P. Queiroz, S. M. Dias, J. M. Duarte, M. M. Carvalho, "Uma Solução Para O Sistema Brasileiro De Coleta De Dados Ambientais Baseada Em Nanossatélites", Holos, Dez, 2018, https://doi.org/10.15628/holos.2018.6307

Appendix C - Enhanced software development process for CubeSats to cope with space radiation faults

Enhanced software development process for CubeSats to cope with space radiation faults

David Paiva
CISUC, University of Coimbra
Coimbra, Portugal
davidpaiva.uc@gmail.com

Raffael Lima
COENE, INPE
Natal, Brazil
raffael.sadite@inpe.br

Manoel Carvalho
COENE, INPE
Natal, Brazil
manoel.carvalho@inpe.br

Fátima Mattiello-Francisco
COEPE, INPE
São José dos Campos, Brazil
fatima.mattiello@inpe.br

Henrique Madeira
CISUC, University of Coimbra
Coimbra, Portugal
henrique@dei.uc.pt

***Abstract* — CubeSats are an established trend in the space industry. The Cubesat standard opens opportunities for rapid and low-cost access to space. The use of COTS components instead of space-hardened hardware greatly reduces the cost of Cubesat-based missions and provides the additional benefit of increasing software functionalities with low power consumption. However, COTS components are not designed for the space environment, making CubeSats sensitive to space radiation. This means that CubeSats need additional software mechanisms to guarantee resilient behavior in the presence of space radiation. Our proposal is that such software implemented fault tolerance mechanisms must be tailored to the specific code running in each CubeSat and the logical way to achieve that is to extend the software development process for CubeSats to include the systematic resilience evaluation of software as part of the CubeSats software lifecycle process.**

This paper proposes a set of structured steps (and tools to support such steps) to enhance the classic software development process used in CubeSats, focusing particularly on the Verification and Validation (V&V) phase. The approach uses fault injection as an integral part of the development environment for CubeSats software and includes three major steps: a) sensitivity evaluation (verification) of software in the presence of faults caused by space radiation, b) strengthen of the software with targeted software implemented fault tolerance (SIFT) mechanisms and c) validation of the effectiveness of the SIFT mechanisms to confirm that the software is immune to space radiation faults. These added steps to the V&V process must be carried out during software development, as well as every time the CubeSat software has an update, or even a minor change, to ensure that the impact of faults caused by space radiation is tolerated by the CubeSat software. The paper demonstrates the proposed approach using three different embedded software running in the EDC (Environment Data Collection) CubeSat board, which is part (payload) of a constellation of satellites being developed by the Brazilian National Institute for Space Research (INPE). EDC use case provides a realistic insight on the effectiveness of the proposed steps. Our results show that the proposed approach can reduce the percentage of silent data corruption (the most problematic failure mode) from the range of 15% to less than 1% and even to 0% in some embedded software, meaning that the CubeSat software becomes immune to space radiation.

Keywords — CubeSats, COTS, software development, verification and validation, soft errors, fault injection, software fault tolerance techniques

I. Introduction

Nowadays, the interest in the development and deployment of CubeSats solutions has become a trend in the space industry. CubeSats are small-satellites built with up to 12 units in the shape of a cube of 10cm edge and weight of 10kg maximum, according to the CubeSat Design Specification (CDS) - a standard (de facto) for mechanical design and interfacing for satellites [1]. In fact, the Cubesat standard strongly reduces cost and development time of space projects, increases accessibility to space, and sustains frequent launches. Cubesat-based projects place emphasis on the use of Commercial-Off-The-Shelf (COTS) components and systems. When compared with space-hardened components - specially designed to withstand the harsh space conditions - COTS present several benefits like low cost, high performance, and low energy consumption, which open opportunities to develop new space technologies and carry out space missions in the fastest and cheapest way.

Despite these advantages, COTS components have not been designed for space applications, which means they are susceptible to transient errors as a result of single event upsets (SEU) caused by space radiation. In fact, errors caused by SEU are established as the major cause of COTS components failures in space [2]. The impact of space radiation could damage COTS components on a permanent basis, but the most common effect is to cause transient faults [2] that may lead the software to crash or to produce erroneous results.

Although CubeSats generally use ordinary hardware COTS components (i.e., components sensitive to space radiation), typical architectures of CubeSats boards [3] include several mechanisms to cope with SEU and faults caused by space radiation. Memory is typically protected through error detection and correction codes, and communication structures also use error detection and correction provided by the communication protocols and associated hardware of the communication links. Memory, in particular, represents a large silicon surface exposed to radiation, which means that protecting memory from transient bit flip errors due to space radiation is mandatory.

Fortunately, the protection of memory and communication channels against transient faults caused by space radiation is relatively easy to achieve at low cost because of the regular nature of such structures. For example, the use of extended Hamming codes [4] to assure single error correction and double error detection in the memory just requires two extra parity bits and is a frequent solution in CubeSat boards. Similarly, the use of communication protocols and techniques such as forward error correction codes [5] are effective in dealing with errors caused by transient faults in communication channels.

The big challenge is to protect the processor(s) of CubeSat boards from the effects of space radiation. Obviously, the use of space-grade processors that resist space radiation is not an option for CubeSats, as the cost of such processors is several orders of magnitude higher than the cost of common COTS processors. But, unfortunately, COTS processors are not immune to space radiation and, at the same time, the complex internal structure of processors does not allow the use of affordable data error detection and correction methods that protect uniform and regular structures such as memories and communication channels. In other words, existing CubeSat boards can deal with transient faults caused by space radiation that affect memory and communication, but the processor represents the major weakness for the reliability of CubeSats.

The obvious solution would be to rely on classic fault-tolerant architectures at the board level [6] to tolerate faults of the COTS processors in CubeSats. But these techniques represent a substantial increase of hardware redundancy, with high negative impact on the board weight and power consumption. For example, the use of duplicated processors in CubeSat boards would require a large amount of additional hardware to deal with the comparison of the two processors, no matter the concrete flavor of fault-tolerant architecture used in the board design. For example, techniques such as lock-step dual processor architectures would require the low-level comparison of the hardware signals of both processors (and, most likely, can only be used if the processors are implemented in FPGAs to have access to the internal processor structure to allow synchronization of signals). Other architectures such as symmetric multiprocessors (i.e., two or more identical processors sharing a single main memory) would also need substantial additional hardware and have negative impact at other levels (e.g., would require a multiprocessor-aware operating system) [6].

Recent research work (PhD thesis of C. Fuchs, December 2019 [7]) proposes a novel on-board-computer architecture for very small satellites (<100kg) capable of achieving high reliability without using radiation hardened semiconductors, through the combined use of hardware and software-implemented fault tolerance techniques [7]. However, in spite of this promising research result from C. Fuchs, to the best of our knowledge, there are no fault-tolerant boards available for CubeSats, especially boards that can cope with transient faults that affect the processor, which are the major threat for the reliability of CubeSats.

The current situation in the space industry is that, in spite of the growing interest in CubSats, this category of miniature satellites is still considered as not adequate for high-priority and critical missions, and the reason is the low reliability of CubSats [8]. Data from 178 launched CubeSats show that the 2-year reliability estimation ranges from 65% to 48% [8]. The detailed analysis of the results presented in [8], concerning the subsystem identified as root cause of the failure, shows that the payload subsystem contributes with modest figures (from 3% to 4%), which make sense in an analysis focused on failures of CubeSat missions with a strong incidence of DOA (dead-on-arrival), where the satellite never achieved a detectable functional state. However, we may speculate that the failure rate in CubeSat payload software could be much higher, especially considering transient failures in the payload software that, apparently, has not been considered in [8].

In this paper we propose a pure software implemented solution that allows us to improve the reliability of existing CubeSats, without requiring any change or extra hardware in the CubeSats boards currently available. Our approach takes into account the fact that the impact of faults caused by space radiation at processor level is highly dependent on the actual software running in the CubeSat, as the error propagation phenomena and the translation of the erroneous behavior caused by faults into critical failure modes depend on the intrinsic characteristics of the code, namely low-level features such as the data structures and code constructs. Abundant fault injection literature shows that depending on the actual code, the effect of faults could be relatively minor or could be devastating. This fact can be attested in many fault injection papers as reported (and condensed) in periodic surveys and fault injection papers (see [10], [11], [12]).

In particular, a project involving NASA JPL and authors of current paper [13] reported results from an injection campaign on a NASA COTS-based payload system for onboard processing of scientific data and shows that the percentages of the different failure modes are quite dependent on the software running in the system at the moment when the faults were injected. This difference reaches up to 45% in some failure modes, particularly in potentially dangerous failure modes such as "silent data corruption", in which the radiation induced faults cause erroneous software results but do not activate any error detection mechanism available in the system.

This persistent dependency of the impact of transient faults on the actual software running in the CubeSats suggests that software implemented solutions must be instantiated at all levels of the software development lifecycle, considering both system software (operating systems, libraries, etc.) and application (payload) code. This is precisely the goal of the present paper that proposes an additional set of steps for the development of software for CubeSats. The contributions of the paper can be summarized as follows:

- Proposes an extension of the software development process for CubSats using fault injection as an integral part of the development setup to **verify** the sensitivity of the software to space radiation induced faults and **validate** subsequent iterations of the software enhanced with software implemented fault tolerance mechanisms.
- Presents the proposed extension as a set of additional steps to the V&V phase. These steps include 1) sensitivity analysis, 2) software enhancing with fault tolerance techniques and 3) validation of the final software resilience. Although this extension is presented and discussed in the paper as part of the traditional waterfall V model, it is largely agnostic concerning the software development process and can be used in agile methodologies as well.
- Applies the proposed methodology to a concrete CubeSat board and shows that the effects of transient faults induced by space radiation can be reduced to nearly zero using the proposed approach.

The structure of the paper is as follows: next section presents a brief state of the art on software development practices for CubeSats; section III presents the proposed approach; section IV describes the use case from the INPE CubeSat, including the application do the proposed approach and discussion of the results; and section V concludes the paper and outlines future work.

II. SOFTWARE DEVELOPMENT PRACTICES FOR CUBESATS

The advent of the CubeSat standard has made the development cycle of small satellite projects (e.g., CubeSats) much faster and cheaper, when compared to larger space missions. The satellite structure, cabling and interfaces have been significantly simplified with the Cubesat standardization, but the complexity of software embedded in the satellite subsystems just increased. Thanks to technological evolution of embedded electronics, memories and satellite processors, the potential for adding functionalities implemented by software has grown. [1313] Subsystems onboard Cubesat-based satellites (platform and payload) have their own software architecture. The challenge lies in the integration of the so-called software-intensive systems (SiS) in a short development time imposed by Cubesat missions [14] [15].

The focus on the concept of interoperability of SiSs aboard spacecraft is not a concern limited to Cubesats. In the last 20 years, the satellite industry evolved from viewing software, mostly developed in house, as an important aspect of the entire spacecraft, to dealing with the integration of SiSs provided by different suppliers. Efforts in the verification and validation process were necessary to support the integration phase with tools and methods, which made the process onerous in time and resources. This is acceptable in the development cycle of traditional satellites but incompatible to Cubesat philosophy, whose project shall be much faster and cheaper [16] .

Moreover, the classic waterfall model usually adopted in the 90´s for space software engineering gave way to incremental approaches and agile methods, which easily cope with the addition of new requirements and more agile methods. Currently, the development of most CubeSat software follows such approaches. The On Board Data Handling (OBDH) typically grows in complexity because new services are required by payloads late in the development time. More pieces of software are developed to interface, control and operate different subsystems. These new features increase the overall complexity of the satellite and also increase the number of possible software defects [14] [15]. Considering that the success of CubeSat missions relies on the perfect operation of the OBDH, it is very useful to mitigate such vulnerabilities without increasing the costs of a V&V process.

Regarding software assurance practices, simulation and testing are the most common activities to verify and validate CubeSats software, according to a survey conducted at NASA Ames Research Center [17]. However, even those activities do not receive due attention on Cubesats projects, as an intensive program of verification and validation cannot be accommodated into the limited budget of such projects. Despite this, according to the same survey [17], an emerging trend relies on the use of model-based design methods due to their capability to automate the creation of detailed software design from high-level graphical inputs, and then use automatic code generation to create the code. Although, this type of modeling/software development requires expertise on the part of the developer of the tool used. The definition of requirements for generating the model are fundamental for the fidelity of this model to the desired implementation. Both the correction of bugs found in the tests and the integration of designed modules must be done at high level, hand-codes are not allowed to guarantee the reliability of the designed model.

Although fault-tolerant software methods are used for run-time monitoring in CubeSats, the use of rigid verification and validation techniques is not a trend in current Cubesats software development due to time and budget constraints of such projects. This includes the crucial verification of possible effects of space radiation induced faults.

III. ENHANCED VERIFICATION AND VALIDATION FOR CUBESATS SOFTWARE DEVELOPMENT

Our proposal focuses on enhancing the verification and validation of CubeSats software through a set of additional steps. These steps are intended to be the least intrusive possible on the software development life cycle used by the companies, space agencies, and other institutions that are developing CubeSats. Since budget and time are constraints that must be considered, expensive software verification and validation activities are impossible to accommodate on such projects. The proposed steps require a fault injection tool, but such tools are readily available at low cost, such as the tool CubeSatFI [18], which can be easily integrated into the software development process of CubeSats without significantly increasing the cost of such projects. Moreover, the proposed approach when used in the early stages of the software development life cycle cannot only find weak points caused by space radiation-induced faults, but also can be useful to find software bugs not discovered during integration testing activities. Fig. 1 illustrates the proposed additional steps. More specifically, our proposal does not change the previous phases of the existing software development process, but simply adds additional V&V steps after the integration test step, which is always part of the process, no matter the flavor of the software development process used by the CubeSat developer. The proposed steps assume that a fault injector tool such as CubeSatFI [18] capable of injecting transient faults similar to the ones caused by SEU is available:

Step 1 - Evaluate the software sensitivity to space radiation: After integration testing the software is subject to a comprehensive fault injection campaign to evaluate the impact of SEU on the CubeSat behavior. Faults are injected in the processor registers of the target board using a random distribution (both in space - registers- and time), since space radiation tends to affect the processor randomly. This will allow us to understand the behavior of the target software in the presence of space radiation that affects the processor of the board where the software is running.

Step 2 - Strengthen the software with tailored software implemented fault tolerance (SIFT) techniques: The results obtained in the previous step must be analyzed and the impact of the faults on the target software should be categorized into failure modes. According to the failure modes obtained, it should be decided if it makes sense to add additional SIFT techniques to the code to avoid failure modes such as **silent data corruption** (erroneous output results with no error detection) or to recover the software after **crash failure modes**. This decision should be taken considering the available resources of the target system and the budget available to implement these techniques . Many SIFT techniques can be used (see, for example, chapter 5 of [19] or the classic Michael Lyu's book [20]), from simple re-execution and voting to self checking software. If the target system has enough resources, it is extremely recommended to add SIFT techniques to increase fault coverage as much as possible. Obviously, we are aware that including additional SIFT techniques in the CubSat software after a first version of the software has been through integration testing could be problematic. For fault masking techniques such as software re-execution and voting [19], [20] this task of adding this technique to existing software is relatively easy. But for other SIFT techniques such as algorithm-based fault tolerance [20] the existing software must be largely refurbished to incorporate the SIFT technique.

Step 3 - Validate the effectiveness of the SIFT techniques: After the software is strengthened with additional SIFT techniques, it must be submitted to regression testing (using a test suite developed in earlier stages of the software development lifecycle) to assure that the functional requirements (and also non-functional requirements such as response time) are still met. The validation of the effectiveness of SIFT is then performed through a fault injection campaign similar to the one run in step 1. That is, the process enters the cycle proposed in Fig. 1 until the desired software resilience in the presence of transient faults is achieved. The objective is to evaluate the effectiveness of SIFT techniques in the mitigation or toleration of transient faults such as the ones induced by space radiation. If the results do not satisfy the targeted reliability.

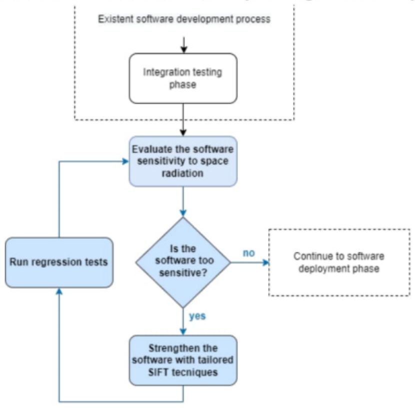

Existent software development process
Integration testing phase
Evaluate the software sensitivity to space radiation
Is the software too sensitive?
no
Continue to software deployment phase
yes
Strengthen the software with tailored SIFT tecniques
Run regression tests

Fig. 1. Enhanced verification and validation steps for cubesats software development process

The proposed steps should be included in the software development process used in the CubeSat development project. If the project follows the classic V-Model, the proposed steps should be included after the integration testing phase (right-side of the V). If the CubeSat project follows an agile process (this is a growing trend in CubeSats development), the proposed steps should be performed each time that the software has a considerable increment. Since the impact of SEU-induced faults depends on the actual software that is running on the CubeSat, every time the software changes, it is crucial to perform the proposed additional V&V steps. In fact, these steps are quite inline with test-driven development (TDD) used in agile development processes, where the software requirements are converted into test cases and each software increment aims to pass the new set of test cases. After the test pass, the code is refactored, and the test suite is run again to assure that no existing functionality is broken. This cycle is repeated for each new functionality. Similarly to TDD, when the CubeSat software has a major or even a minor change, the proposed V&V steps should be executed to evaluate the resilience against SEU-induced faults, and the software is considered fully developed when it meets the safety and dependable requirements to tolerate space radiation.

An important aspect for the actual application of the proposed approach is the availability of fault injector tools such as CubeSatFI [18] as part of the software development environment to allow the execution of fault injection campaigns in an easy and automatic way (and at low cost).

IV. USE CASE: SBCDA CUBESAT

This section presents a use case of the proposed enhanced V&V approach using the Environmental Data Collector (EDC) [21], a Cubesat payload board for the Brazilian Environmental Data Collection System (SBCDA). Is not worth mentioning that this payload is going to be used in all the future CubeSats from the CONASAT-project [22].

A. Cubesat CONASAT-1

The Brazilian Environmental Data Collection System (SBCDA) has operated since 1993 after the launch of the SCD-1 data collection satellite. The SBCDA aims to collect data such as wind speed, rainfall and temperature indices, wildlife observation, among others. In addition to the SCD-1, the system is currently supported by the SCD-2, CBERS-4, and CBERS-4A satellites (much larger than CubeSats), which carry on board an analog transponder that relays the signals from the data collection platforms (DCP) to the receiving stations (RS) on the ground.

The EDC is a new payload developed to meet the demand for a signal receiver CubeSat-compatible to provide onboard signal processing. The EDC design uses COTS components which gives it a low-cost, however, it makes the EDC system less reliable than the analog transponder. By offering onboard processing, the EDC is capable of expanding the SBCDA systems service coverage. The development of this payload is convergent with the CONASAT project, which aims to launch a constellation of low-cost nanosatellites to renew and expand the Brazilian data collection system.

CONASAT-1 is the first satellite in a constellation of low-cost nanosatellites based on COTS components. CONASAT-1 is based on a CubeSat platform with a size of 1U, i.e., this satellite has a cubic shape with edges of 10 centimeters. CONASAT-1 uses the hardware platform developed by the EnduroSat company. The CONASAT project team is responsible for developing the flight software for the onboard computer (OBC), and together with the EDC team, carrying out the integration of the satellite with this payload. In addition to the OBC and the EDC, the CONASAT-1 hardware architecture comprises two UHF antennas, a UHF transceiver, an electrical power system (EPS), and a battery pack. Fig. 2 illustrates the hardware architecture in block diagram level described.

The UHF antenna - Payload is dedicated to receiving signals from DCPs and is connected to the EDC, while the UHF antenna - TMTC is used for communication with the RSs and is connected to the UHF transceiver. The UHF transceiver is the subsystem responsible for receiving and transmitting the telecommand (TC) and telemetry (TM), respectively. The EPS subsystem supplies power to the entire platform through six solar panels and several voltage converters. The platform also has a battery pack with a capacity of 10.2 Wh. The OBC is responsible for configuring, controlling, and commanding the operation of the subsystems. The telecommands received from an RS are decoded in the OBC to control the subsystems onboard the satellite. The OBC is also responsible for monitoring the overall health of the satellite. Health assessment can be performed in several ways, depending on the subsystem being assessed. Telemetry sensors are used to verify that the parameters of a given subsystem are acceptable (such as temperature or voltage level). Telemetry data collected from each subsystem is also stored for transmission to an RS. The OBC acts as the I2C bus master for transmitting commands to the EPS subsystems, UHF antennas, and UHF transceiver. The flight software implements in the OBC a routine of commands and requests to control the data processed by the EDC. This is performed through a UART communication interface. With the payload data in hand, the OBC uses the USART interface to transfer it to the UHF transceiver. The UHF transceiver transmits through the UHF antenna TMTC at the frequency of 462 MHz. While the beacons are transmitted by the same antenna at the frequency of 435 MHz. The UHF transceiver is configured for a baud rate of 9600 bits per second.

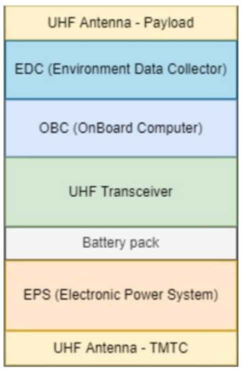


Fig. 2. Block diagram of CONASAT-1 Hardware Architecture Overview

B. Application of the proposed approach

Aiming to demonstrate the effectiveness of the proposed enhanced steps in the software development process for CubeSats, we decided to deploy three different embedded software on the EDC board that will be used on CONASAT-1 CubeSat. The three embedded software applications are:

- **Matrices**: it is a program that computes the result of the multiplication of two matrices 30 times and at the end of each run, calculates a cyclic redundancy check (CRC) for the result of the multiplication. After the 30 runs, calculate a final CRC of the 30 CRCs previously calculated. In our experiment, the program uses two 30x30 integer matrices.
- **PI**: Computes the value of π using the Leibniz formula. In our experiment, the program computed the π using 60000 terms.
- **Fibonacci:** it is a recursive program that computes the sequence of Fibonacci and sums the calculated elements. In our experiment, the program computed the sum of the first 30 elements of the Fibonacci sequence.

It is worth mentioning that these payload software applications do not correspond to software payload that is going to fly in future CubeSats from INPE. However, they run on top of the real software running on the EDC board, namely the FreeRTOS operating system and all the software needed for exchanging messages between EDC board and the OBC board. In practice, the three payload software applications developed have been designed to impose a considerable processing load to the EDC board and reproduce the case of payload software that takes a considerable amount of time to execute.

The fault injection (for the software sensitivity evaluation step) was performed using CubeSatFI [18], a fault injection tool that takes advantage of the modern features of the actual microcontrollers by injecting faults in a fully automated way through the JTAG interface. The tool offers the possibility of designing an entire experiment, choosing the specific target that we want to affect, as well as the moment that we want the fault to be injected.

The target of the injected faults are the registers of the processor of the EDC board. As discussed before (Introduction), the processor is the main weak point for the reliability of CubeSats boards in the presence of space radiation. In the ODC board, memory is protected with single error correction and double error detection parity codes and the message exchange between the EDC and OBC boards is also protected with error detection mechanisms.

The fault injection campaigns consist of 2000 faults. All faults are single bit-flips faults, as this model is broadly accepted as a realistic simulation of SEU faults. With that in mind, the register affected by each fault was selected randomly (among all the processor registers) and the bit of the register affected by the faults was also selected at random. The trigger of each fault is also randomly defined within an injection window (i.e., within a time interval defined by the tester). The injection window interval was defined as between 2 and 4 seconds after resetting the CubeSat to assure that each fault is injected in the system without having the effects of previous faults.

At the end of the injection window, the target system will be running for a period of 26 seconds to collect data for further analysis of the effects of the fault. The results produced by each software application are sent to the host computer that is executing the CubeSatFI through the UART interface of the EDC payload board. The results of the campaigns are saved in a file to further analysis.

These campaigns with randomly injected faults (both in the register space and in time) are appropriated to emulate the effects of transient faults caused by SEU, as space radiation tends to affect the processor in a random way. We decided to keep the single bit-flip model and not to include faults injected in multiple bits of registers because these multiple bits faults (caused by space radiation bursts) tend to cause drastic impact on the software and are easy to detect, and consequently are easy to handle.

It is worth noting that due to the random nature of the injection process, the injected faults may affect either the payload software application or the EDC software, namely the FreeRTOS operating system and the software used for exchanging messages between EDC board and the OBC board, which represents a realistic scenario for SEU faults.

The process was applied following the three additional V&V steps of the proposed approach:

1. Sensitivity evaluation by applying the fault injection campaigns (one campaign for each payload software) to the original software (i.e., without specific SIFT techniques, unless some error detection techniques such as watchdog timer available in the EDC board).
2. Strengthen of the payload software with a simple SIFT technique that consists of r-execution of the payload software and voting of the results.
3. Validation of the effectiveness of the SIFT technique through the fault injection campaigns.

Next subsection presents and discusses the results.

C. Discussion

Fig. 3 shows the general impact of faults in the three scenarios of running the payload applications on the EDC board, corresponding to step 1, before the implementation of SIFTtechniques and considering the 2000 faults injected in each case.

Based on the results obtained from the experiment the failures were classified according to the following failure modes:

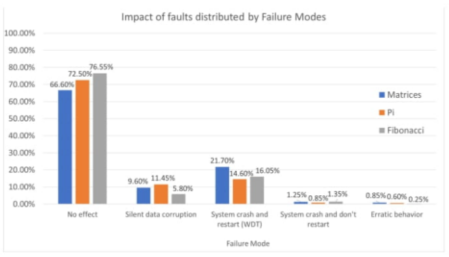


Fig. 3. Impact of faults distributed by failure modes

- **No effect**: The fault had no visible impact on the system, which means that the CubeSat continues to work normally and the expected results are received by the onboard computer.
- **Silent data corruption (SDC)**: The fault had no visible impact on the system. However, the results sent are incorrect.
- **System crash and restart (WDT)**: The system crashes and the watchdog timer (WDT) is activated. After the WDT activation, the system is restarted and goes back to working properly.
- **System crash and do not restart**: The system crashes, remaining in that failure mode without WDT activation.
- **Erratic behavior**: The system sends wrong information repeatedly that can be detected by the mechanisms available in the CubeSat.

Considering the failure modes presented above, "silent data corruption (SDC)" is the worst of all and represents a serious risk since the faults that lead to this failure mode are impossible to detect. The code control flow is not affected, instead, the system produces an erroneous result impossible to be detected, which means that SIFT techniques must be considered to avoid the serious failure caused by these faults. In contrast, when the system crashes and it is detected by the WDT, the system is restarted and back again to work as expected, ensuring system reliability. Likewise, the "system crash and do not restart" failure mode can be easily managed with external mechanisms such as a Heartbeat system that must receive a signal periodically from the CubeSat payload system. If the signal is not received means that the system is in a blocked situation after a crash and must be forced to restart. Finally, the faults that lead to an "erratic behavior" can be easily detected by the onboard computer that should restart the EDC board.

The high percentage of faults that have no impact ("No effect" failure mode) in three payload application scenarios is quite normal and corroborates previous fault injection experiments reported in the literature (e.g., [9]). This percentage varies between 66% and 76% depending on the software under test. The reason for such results can be justified by the intrinsic redundancy existing in computer systems and software.

Analyzing in more detail the failure mode distribution for the different processor registers, Fig. 4 shows that some processor registers are not affected at all by the injected faults. The reason is because these registers (e.g., R5, R6, R8, R9, R10, R11, R12) are not used by the code. Of course, these situations vary according to the actual software that is being executed in the CubeSat. Furthermore, the way the software uses the available resources of the processor is defined by the C compiler switches during the compilation phase (the EDC firmware and the codes used in this experiment are developed in the C language). In fact, the result of fault injection campaigns can be quite different if the code is compiled with different compilation switches, as this can influence the behavior and performance of the software in execution. Also, the fact that a big number of registers are normally not used by the compiler opens some possibilities to implement extra SIFT mechanisms.

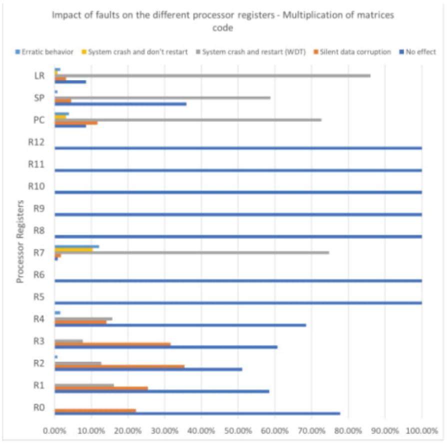


Fig. 4. Impact of faults on the different processor registers - Multiplication of matrices code

To minimize the impact of the transient faults caused by space radiation, we added the "plain-vanilla" version of the SIFT technique known as re-execution and voting [19], [20]. In practice, the code is executed twice and the result is voted in order to decide if it is trustable. If the two results differ, the code is re-executed and compared with the two previous results. In the end, if the third run does not match either of the previous two, a message of error is sent to the output. In contrast, if the result matches one of the first two, it means that one execution was affected by the space radiation but the others can be considered trustable.

Fig. 5 shows the distribution of the faults according to the different failure modes in each processor register after the application of the software fault tolerance technique explained above. **The results show that the re-execution and the voter can almost eliminate the impact of faults that cause SDC** on the general registers of the processor that are being used by the code (e.g., R0, R1, R2, R3, R4, R7). On R0 and R7 the silent data corruption is totally tolerated, turning these registers immune to this type of failure mode. Also in the R0, the simple technique of re-executing and voting turns this register immune to space radiation as this register presents a percentage of 100% of "No effect". Positively, in the other registers the percentage of "No effect" exceeds 80%, ensuring the reliability of the CubeSat against space radiation.

Also in Fig. 5 we can see that the simple software fault tolerance technique used in the multiplication of matrices code just mitigates the effect of faults that leads to SDC on the special registers of the processor (e.g., PC, SP, LR). Looking at the other failure modes, we can see that the results did not change too much. This phenomenon is expected since these registers are special registers of the processor, which means that any fault injected into one of these registers can lead the system to an incoherent behavior or even block the entire system. To strengthen these registers and increase the percentage of "No effect", more sophisticated error detection mechanisms must be added to the software under development (e.g., self-checking routines).

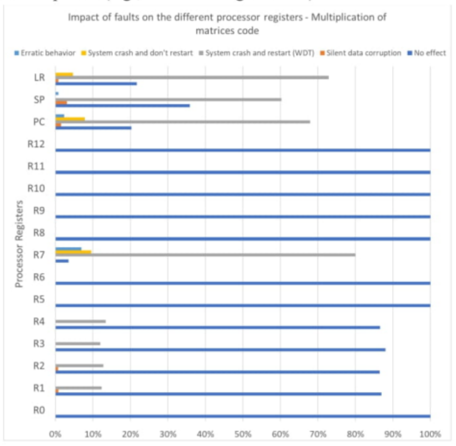


Fig. 5. Impact of faults on the different processor registers - Multiplication of matrices code

Fig. 6 compares the impact of faults considering the three application scenarios before and after the payload applications have been strengthened with the re-execution and voting SIFT technique. Additionally, the analysis considers only the faults that affected the registers that are being used by the software under test (since faults injected in registers that are not used always lead to "No effect" failure mode). As mentioned before, the impact of transient faults is dependent on the actual software that runs on top of the CubeSat, and looking at the two graphics in Fig. 6 we can see that the sensibility to space radiation varies according to the software running on the EDC board. The multiplication of matrices is the most sensible code, presenting a percentage of "No effect" that almost reaches 42% without any software fault tolerance technique and 59% with a simple re-execution and votation. However, the simple software fault tolerance technique increases the resistance of all the software and in the particular case of the multiplication matrices code, increases the resistance to "silent data corruption" by more than 17%.

Focusing on the effectiveness of the re-execution and voter implemented on the three software, we can conclude that even the most simple software fault tolerance technique can increase the reliability of the CubeSat. The results presented in Fig. 6 show that the "silent data corruption" failure mode becomes residual, after the introduction of the re-execution and voter, on all the software under test. In fact, the percentage of "silent data corruption" on the software that does the calculation of the PI is 0%, proving that such software is immune to faults that lead to this type of failure mode. Nevertheless, in the other two codes, we still have some occurrences of SDC, since the technique applied is two simple and even the voter can be affected. However, with a more sophisticated (e.g., duplicated voter) the results can be even better.

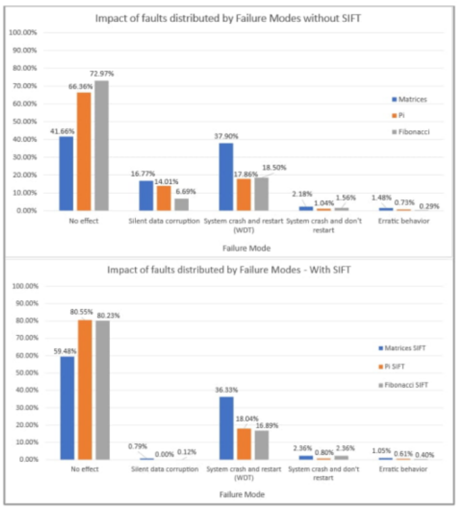


Fig. 6. Comparison of the impact of faults distributed by failure modes on the all the software tested before and after being strengthened with SIFT techniques

In addition to the software fault tolerance added to the software, the EDC board includes a watchdog timer (WDT). This error detection mechanism is present in all CubeSats boards and plays a very important role in detecting system crashes. Looking at Fig. 6, we can see that most failures that lead to system crashes are detected by the watchdog timer (i.e., the failures are classified as "System crash and restart (WDT)"). This means that after the system crash is detected by the WDT, the system is restarted and back to work as expected, assuring the availability of the system. Taking advantage of the WDT together with the software fault tolerance technique added to the embedded software, it is possible to make the CubeSats almost immune to SEU faults.

This simple example proves the effectiveness that the software fault tolerance techniques have in the prevention of failures caused by space radiation.

V. Conclusions and Future Work

This paper proposes an enhanced software development process for CubeSats to cope with space radiation faults. Cubesats open up opportunities for fast and cheap access to space taking advantage of the use of commercial off-the-shelf (COTS) components. However, these components are susceptible to hardware transient faults caused by space radiation, and the CubeSats software projects do not adopt rigid verification and validation activities to manage the impact of space radiation on COTS components.

The proposed solution intends to be easy to adopt on the software development life cycle used by companies, space agencies, and other institutions that are developing CubeSats. In fact, the proposed steps can be applied to projects that follow a classic development process, like V-Model, as well as to agile processes. The key idea is to use fault injection (using available fault injectors, such as CubeSatFI) to emulate transient faults caused by space radiation and analyze their impact on CubeSat software. The impact caused by the injected faults should be categorized into failure modes. To mitigate or even tolerate critical failure modes (such as silent data corruption), tailored software implementing fault tolerance (SIFT) techniques must be added to the software under test. Hence, the software must again be submitted to a fault injection campaign aiming to evaluate the effectiveness of the SIFT techniques. Following these steps, regression tests should be run to assure that the software functionalities are still working as expected. These added steps must be performed every time the software has an update, or even a minor change, to evaluate the CubeSat resistance capability against space radiation.

In short, we can summarize the proposed solution in the following steps: 1) Evaluation of the software sensitivity to space radiation; 2) Strengthen the software with tailored software implemented fault tolerance (SIFT) techniques; 3) Validate the effectiveness of the SIFT techniques.

The paper also presents a use case of the application of the proposed enhanced verification and validation steps proposed using the Environmental Data Collector (EDC), a Cubesat payload board for the Brazilian Environmental Data Collection System (SBCDA) that will be used on all CubeSats from the CONASAT-project. We deployed three embedded software on the EDC board and we submitted them to an intensive fault injection campaign aiming to evaluate their sensibility to space radiation.

Results show that the impact caused by the faults is different according to the software under test (which is normal, as the error propagation phenomena and the translation of the erroneous behavior caused by faults into critical failure modes depend on the intrinsic characteristics of the code). Besides, all software tested present a considerably high percentage of "silent data corruption", which represents the worst failure mode caused by space radiation-induced faults, as the fault had no apparent impact on the system but, the results produced are wrong. By applying a SIFT re-execution and voting technique, we can reduce the occurrence of this failure mode to residual values. In fact, in one of the payload applications tested, this failure mode is totally avoided, turning the CubeSat immune to space radiation for this critical failure mode.

Detection faults mechanisms (e.g, watchdog timers, and others) and SIFT techniques can dramatically increase CubeSat's reliability without requiring any change in the current CubeSats boards, making the proposed enhanced software development process a promising approach in the development of reliable solutions for CubeSats missions.

ACKNOWLEDGMENT

This work was supported in part by the Grant CISUC-UID/CEC/00326/2020, funded in part by the European Social Fund, through the Regional Operational Program Centro 2020, and supported in part by the H2020, Marie-Curie project ADVANCE ("Addressing Verification and Validation Challenges in Future Cyber-Physical Systems"), project funded by the VALU3S ("Verification and Validation of Automated Systems' Safety and Security") project, supported by the European Commission, and in part by European Leadership (ECSEL) Joint Undertaking (JU) under Grant 876852, and in part by the JU from the European Union's Horizon 2020 Research and Innovation Programme.

REFERENCES

[1] "CubeSat Design Specification (1U - 12U) REV 14", CP-CDS-R14.

[2] R. Ecoffet, "Spacecraft Anomalies Associated with Radiation Effects", in RADECS 2013 Short Course Proceedings, Chap. VIII, 2013.

[3] F. Davoli, C. Kourogiorgas, M. Marchese, A. Panagopoulos, F. Patrone, "Small satellites and CubeSats: Survey of structures, architectures, and protocols", Int. Journal of Satellite Communications and Networking, September 2018.

[4] T. K. Moon, "Error Correction Coding. New Jersey: John Wiley & Sons", ISBN 978-0-471-64800-0, 2005.

[5] Z. Yuan and X. Zhao, "Introduction of forward error correction and its application," 2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet), 2012.

[6] D. Sorin, "Fault tolerant computer architecture." *Synthesis Lectures on* Computer Architecture 4.1 (2009): 1-104. 2009.

[7] C. M. Fuchs, "Fault-tolerant satellite computing with modern semiconductors," Ph.D. dissertation, Leiden University, 2019.

[8] M. Langer and J. Bouwmeester, "Reliability of CubeSats-statistical data, developers' beliefs and the way forward," in AIAA/USU Conference on Small Satellites (SmallSat), 2016.

[9] H. Madeira, R.Some, F. Moreira, D. Costa, D. Rennels, "Experimental evaluation of a COTS system for space applications", Int. Conference on Dependable Systems and Networks, DSN-2002, Bethesda, Maryland, USA, 2002.

[10] M-C Hsueh, T. K. Tsai and R. K. Iyer, "Fault injection techniques and tools," in Computer, vol. 30, no. 4, pp. 75-82, April 1997.

[11] R. Natella, D. Cotroneo, H. Madeira, "Assessing Dependability with Software Fault Injection: A Survey", ACM Computing Surveys 48 (3), 2016.

[12] R. Barbosa, N. Silva, J. Duraes, H. Madeira, "Verification and validation of (real time) COTS products using fault injection techniques", Commercial-off-the-Shelf (COTS)-Based Software Systems, ICCBSS'07, 39, 2007.

[13] R. Twiggs, "Origin of cubesat," Small Satellites: Past, Present, Future, Eds: Helvajian H., Janson SW, The Aerosp Press, California, 2008

[14] C. Batista, A. Weller, E. Martins, and F. Mattiello-Francisco, "Towards increasing nanosatellite subsystem robustness," Acta Astronautica, vol. 156, pp. 187–196, 2019

[15] D. Almeida and F. Mattiello-Francisco, "Modeling of the interoperability between on-board computer and payloads of the nanosat-br2 with support of the uppaal tool," in 1st IAA Latin American Symp. on Small Satellites. Colombia, 2017

[16] C. Batista, T. Basso, F. Mattiello-Francisco and R. Moraes, "Impacts of the Space Technology Evolution in the V&V of Embedded Software-Intensive Systems" in The 2020 International Conference on Computational Science and Computational Intelligence (CSCI´20: December 16-18, 2020 Las Vegas, USA: Session: Software Engineering Research and Practice

[17] S. A. Jacklin, "Survey of Verification and Validation Techniques for Small Satellite Software Development", NASA Ames Research Center, 2015 Space Tech Expo Conference, May 19-21, 2015.

[18] D. Paiva, J. M. Duarte, R. Lima, M. Carvalho, F. Mattiello-Francisco and H. Madeira, "Fault injection platform for affordable verification and validation of CubeSats software", 10th Latin-American Symp. on Dependable Computing (LADC), 2021.

[19] I. Koren and C. Krishna, "Fault-Tolerant Systems", Elsevier, 2nd Edition – Sept. 2020.

[20] M. R. Lyu, "Software Fault Tolerance", John Wiley & Sons Ltd, 1st Edition 1995.

[21] INPE. "Environmental Data Collector (EDC)". INPE, July 5, 2021. http://www.inpe.br/crn/projetos/edc.php Accessed on: April 10, 2022.

[22] K. P. Queiroz, S. M. Dias, J. M. Duarte, M. M. Carvalho, "Uma Solução Para O Sistema Brasileiro De Coleta De Dados Ambientais Baseada Em Nanossatélites", Holos, Dez, 2018.