SOFTWARE CERTIFICATION AS A LIMIT ON LIABILITY: THE CASE OF CUBESAT OPERATIONS

Marco Crepaldi, Ross Horne, and Sjouke Mauw [*]

Abstract. CubeSats and constellations of CubeSats present new opportunities for low cost access and use of space. As this new technology proliferates, so do risks of in-orbit conjunctions. This chapter will first examine the current status of collision liability under the current legal framework. It will then analyse to what extent CubeSat software engineering practices may influence collision risk. Based on this evidence, the chapter will explore how heightened levels of software certification, with a focus on readily available certified real-time operating systems, could limit the liability of CubeSat operators and launching States.

INTRODUCTION

This interdisciplinary chapter adopts perspectives from space law and software engineering. To see how these remote disciplines impact each other, it is necessary to begin by explaining our argument in such a way that legal experts can understand why they should be aware of software engineering professional standards. Correspondingly, basic legal notions are introduced to provide software engineers with the legal context required to confirm our reasoning. Such interactions are essential for strengthening the transnational interdisciplinary networks of governments and professional bodies that influence space policy[1]. Moreover, this work appeals to space program managers, policy makers, and regulators due to its implications.

Under international space law, launching States are liable for damages caused by space activities – both public and private – therefore, oversight of space missions falls under States' responsibility (per Article VI of the Outer Space Treaty and Articles II and III of the Liability Convention).

While traditionally space missions used highly dependable software, the recent surge in smaller and cheaper missions changes this perspective. More

---

[*] Computer Science Department, University of Luxembourg. We thank Stanislav Dashevskyi for conducting the empirical software engineering experiments.

[1] Andrea Hamann and Hélène Ruiz Fabri. *Transnational networks and constitutionalism*. In: International Journal of Constitutional Law 6.3-4 (2008), pp. 481–508.

precisely, a recent class of space objects, CubeSats,[2] use software that is not highly dependable. If the increasing rate of deployment of CubeSats does not slow down – as seems to be the case – then the quality of the software on board becomes an important issue to address...

...or does it? From the perspective of the satellite operators, if costs are low, then launching large constellations of satellites of which a significant percentage is expected to fail is rational. However, this argument is unlikely to hold from the perspective of launching States that are liable for space activities on the basis outlined above. This is the case if States might be considered at fault when licensing spacecraft running software that is not highly dependable, which would make them responsible for subsequent damages in outer space to persons or property of another operator.

In recent years, risks associated with space missions have increased significantly due to the surge in the number of launches. Among such risks one finds orbital conjunctions, the increase in space debris, and the degradation of the space environment. Successful strategies for managing risks in the medium to long-term, assuming the trend continues as projected, require functional spacecrafts to actively cooperate with tracking and to implement manoeuvres for both collision avoidance and responsible decommissioning. Indeed, the ISO standard for Cube satellites requires (in Clause 5.6.1[3]) that CubeSat mission design and hardware shall be in accordance with the ISO standard for limiting orbital debris[4]. However, granted that standards are not mandatory, we argue that it is possible to interpret current international space law to argue that a duty to ensure a certain degree of dependability of critical software systems already exists.

The fundamental problem when resolving disputes concerning liability in the event of conjunctions in orbit is that the precise cause of collisions is unlikely to be uncovered. There will be little data to witness the event and physical evidence is prohibitively expensive to obtain. However, we expect, as with the historical collision between Iridium-33 and Kosmos-2251 in 2009, that collisions are likely to occur in scenarios where one of the satellites was already in an inactive state prior to the collision. In the aforementioned collision, Kosmos-2251 was a decommissioned soviet satellite that had been

---

[2] James Cutler, Greg Hutchins, and Robert Twiggs. *OPAL: Smaller, Simpler, and Just Plain Luckier*. In: Proceedings of the 14th AIAA/USU Conference on Small Satellites, Logan UT, August 21-24. SSC-VII-4. 2000. url: http://www.space.aau.dk/cubesat/documents/ pdf-docs-from-net/SmallSat2000-Opal.pdf.

[3] *Space systems — Cube satellites*. Tech. rep. 19990. ISO, 2017.

[4] *Space systems — Space debris mitigation requirements*. Tech. rep. 24113. ISO, 2019.

left in orbit. Consequently, more questions were asked concerning the responsibility of Russia, a successor of the launching state of Kosmos-2251, than of the U.S. state, the launching state of the Iridium spacecraft.[5]

An inactive satellite cannot engage in decommissioning, collision avoidance manoeuvres, and cannot actively cooperate with tracking, thereby it poses a hazard to other spacecrafts. To avoid inactivity due to failure, launching States should set baseline engineering requirements for a critical core of the system, sufficient to operate a communication channel (which involves critical components managing power, stabilisation, and communication). This critical core consists of hardware and software. In this work, we focus on the most crucial software component which is typically a real-time operating system (RTOS) which manages both the hardware resources and software processes of critical components.

While there exist ISO standards which can guide the use of hardware in CubeSats[6], baseline standards for software have not yet been emphasised. This paper addressed this gap by analysing RTOS projects commonly used today in CubeSats and demonstrating that even a lightweight empirical evaluation of these projects indicates that commonly deployed RTOS software – most notably Amazon's FreeRTOS – cannot be considered to be sufficiently dependable. The precise degree of dependability varies according to the risks and stakes of each space missions, and it falls under the competence of launching States. It is clear that demanding compliance with full avionics standards that typically apply to spacecraft would be undesirable, since it would result in a surge in the costs for CubeSat operators. Nonetheless, improvements on current practices can be made. In the next pages we show that at least one open RTOS project – namely seL4[7] – follows dependability principles in software engineering.

On this basis, we conclude that launching States should consider including in their registration processes measures to ensure that the critical

---

[5] Ram S. Jakhu. *Iridium-Cosmos collision and its implications for space operations*. In: Yearbook on Space Policy 2008/2009: Setting New Trends. Ed. by Kai-Uwe Schrogl et al. Springer, 2010, pp. 254–275. doi:10.1007/978-3-7091-0318-0_10; Alexander F Cohen. *Cosmos 954 and the International Law of Satellite Accidents*. In: Yale J. Int'l L. 10 (1984), p. 78.

[6] *Space systems — Design qualification and acceptance tests of small spacecraft and units*. Tech. rep. 19683. ISO, 2017, p. 85.

[7] Gerwin Klein et al. *SeL4: Formal Verification of an OS Kernel*. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, 207–220. doi: 10.1145/1629575.1629596.

software deployed on CubeSats does not excessively increase the risk associated with the operations of these space objects.

The above argument for launching States to manage risks associated with software in CubeSats is developed and substantiated as follows. Section I starts by considering the recent surge in small satellite projects, by providing a primer on the liability for space activities under international space law, and explaining the relevance of the role of real-time operating systems on board CubeSats. Then, Section II provides the results of our empirical analysis on three open-source real-time operating systems, namely, FreeRTOS, Kubos, and eCos currently deployed on CubeSats. Our results are compared to the seL4 system, which has a formally verified, hence highly dependable core. Section III, consequently, considers the implications of poor software engineering practices from the perspective of CubeSats' operations and on the attribution of fault in the case of damage caused by CubeSats. Later, Section IV suggests possible mitigation strategies.

I. BACKGROUND: CONNECTING SOFTWARE ENGINEERING WITH FAULT AND COLLISION LIABILITY

The following sections introduce the relevant background notions for the current argument, expanding on our introductory remarks. Accordingly, in Section I.A, we provide evidence of the increasing exposure of States to liability stemming from in-orbit conjunctions by observing trends of space missions and the challenges associated with the increase in the number of small satellites. In Section I.B, a primer on the legal framework regulating the liability for space activities is offered drawing attention to relevant norms in the Outer Space Treaty and Liability Convention. In particular, we connect the liabilities of launching States for in-orbit conjunctions with the notion of fault. In Section I.C, we emphasise that the quality of engineering practices can impact the likely cause of fault, and emphasise that perhaps the most critical software component not currently covered by engineering recommendations is the real-time operating system (RTOS), thereby arguing that the reliability of the RTOS should be brought into consideration.

A. Current trends: increase risk of conjunctions

This section highlights the current trend in the increase of small satellite missions and provides essential information on a specific class of nanosatellites, that is, CubeSats. It aims to show that the trend in CubeSat

missions is unlikely to slow down and, consequently, concerns related to the number or space objects in crowed orbits – such as LEO – will increase in the medium and long-term.

A new era for small satellites has begun. While it is true that the first missions were small – for instance, Sputnik 1 of 1957 was about the size of a beach ball (58 cm in diameter) and weighed only 83.6 kg – the rapid increase in the number of smaller-than-usual space objects is a recent phenomenon. Small satellites (below 500 kg) are divided into several categories according to their mass, for example, microsatellites are considered to be 10-100 kg while a femtosatellite's mass is 10-100 g. The explanation for the recent surge in small satellite missions is manifold. Primarily, these spacecrafts tend to be cheaper, and, therefore expendable. The small size also allows small satellites to be secondary or even tertiary payloads on launch missions, thereby decreasing costs. Analogously, recent technological advances allow for reducing size while preserving complex functionalities.

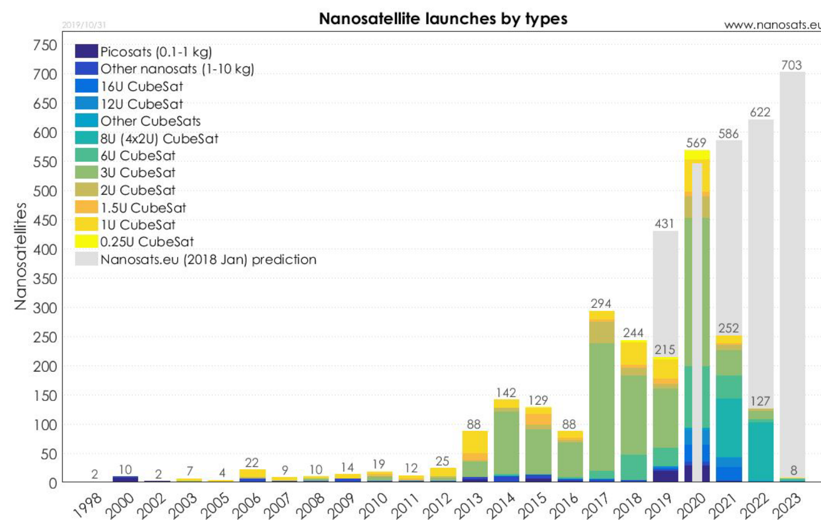

Figure 1: Nanosatellites Launches with forecast, and Cubesats types.[8]

This work focuses on a specific type of small satellites, namely CubeSats. As the name suggests, the core version, known as 1U, of a CubeSat measures 10 x 10 x 10 cm. Multiple configuration are possible ranging between 1 and 40 kg according to the CubeSat standard[9]. The standardization allows for

[8] Data retrieved from https://www.nanosats.eu/database.

[9] A. Mehrparvar. *CubeSat Design Specification*. In: The CubeSat Program, Cal Poly SLO (2014).

smaller and larger satellites made of fraction or multiple of the base unit U, for example, 1.5U, 3U, and 6U CubeSats have been deployed, and may be in the range of 0.25U to 27U.[10] While CubeSats were initially developed for educational purposes, scientific, commercial, and military deployment of these satellites has also occurred. Standardization is also responsible for the lower costs of deployments; for example, a CubeSat mission can cost as little as 50.000 USD.[11] Lower costs and standardization enable broader access to outer space (mainly LEO orbit), which is a welcome development. For example, it enables developing countries to launch their first space objects. Figure 1 illustrates the increasing trend in the launch of nanosatellites with a focus on CubeSat missions.[12]

The growth in the number of small satellites raises several challenges. Most of these challenges are common to all small satellites. Among these challenges one finds, the increase of space debris in crowded LEO, the lack of legal sources that deal with small satellites along with qualification issues, and the dubious status of the future mega-constellations comprising thousands of small satellites.[13] Think, for example, of the Starlink constellation for which SpaceX in 2019 filed for approval of 30,000 more units in addition to the already approved 12,000.[14]

Of course, issues connected to space debris, are common to all space activities.[15] That said, our attention is placed on an issue that is specific to CubeSats, which are designed and deployed quickly on a small budget, hence have reliability issues, such as the dependability of the real-time operating systems used for their control software. Our argument is that, under international space law, launching States have a duty to oversee space missions so that failure to provide adequate guarantees for critical components such as the RTOS on board of spacecrafts is relevant for the regime of the liability for space activities. Therefore, launching States might be considered liable (explored further in Section III) for the damages caused

---

[10] Entry of the Nanosats Database: https://www.nanosats.eu/cubesat

[11] Kiran Krishan Nair. *Small Satellites and Sustainable Development: Solutions in International Space Law.* Springer, 2019. isbn: 3030186202

[12] Robyn M Millan et al. *Small satellites for space science*. In: Advances in space research (2019). issn: 0273-1177.

[13] Mark Matney, Andrew Vavrin, and Alyssa Manis. *Effects of CubeSat Deployments in Low-Earth Orbit.* 2017; Alan Shaw and Peter Rosher. "Micro satellites: the smaller the satellites, the bigger the challenges?" In: Air and Space Law 41.4 (2016), pp. 311–328. issn: 0927-3379.

[14] Article in Space News, October 15, 2019: https://spacenews.com/spacex-submits-paperwork-for-30000-more-starlink-satellites/

[15] Paul B Larsen. *Space law: A treatise.* Ashgate, 2009. isbn: 0754692426.

by CubeSats in LEO. On this basis, it is necessary to digress and describe, albeit briefly, the current international legal framework governing the liability for space activities.

B. The liability for space activities: a primer

This subsection provides a primer on the provisions governing the liability for space activities.[16] It is uncontroversial that CubeSats are to be regarded as space objects and that, consequently, the relevant sources of international law apply. There are two treaties relevant to our subject, namely, the Treaty on Principles Governing the Activities of States in the Exploration and Use of Outer Space, including the Moon and Other Celestial Bodies of 1967 (henceforth the Outer Space Treaty) and the Convention on International Liability for Damage Caused by Space Objects of 1972 (henceforth the Liability Convention). The relevant norms are Articles VI and VII of the Outer Space Treaty and Article II of the Liability Convention. We examine each one in turn.

The Outer Space Treaty. Article VI of the Outer Space Treaty establishes a general responsibility of the Parties to the Treaty with regard to space activities. More precisely, Article VI states the following:

> "State Parties to the Treaty shall bear international responsibility for national activities in outer space, including the Moon and other celestial bodies, whether such activities are carried on by governmental agencies or by non-governmental entities, and for assuring that national activities are carried out in conformity with the provision set forth in the present Treaty."

This disposition establishes international responsibility of States for the activities carried out in space both by governmental agencies and non-governmental entities, including, private entities and universities.

Article VII of the Outer Space Treaty deals directly with liability; however, it has been noted that the distinction between responsibility and liability is only found in the English version of the treaty[17]. This norm clarifies that liability for space activities falls jointly on several States involved in space activities:

---

[16] George Anthony Long. *Small Satellites and Liability Associated with Space Traffic Situational Awareness.* Conference Paper. 2014.

[17] Ram S Jakhu and Joseph N Pelton. *Small satellites and their regulation.* Vol. 3. Springer, 2014.

> "Each State Party to the Treaty that launches or procures the launching of an object into outer space, including the Moon and other celestial bodies, and each State Party from whose territory or facility an object is launched, is internationally liable for damage to another State Party to the Treaty or to its natural or juridical persons by such object or its component parts on the Earth, in air space or in outer space, including the Moon and other celestial bodies."

Several interpretative issues need not concern us, such as what constitutes damage, what precisely constitutes a space object and the standard of proof required to establish causation.[18] It is also important to stress that the principle of state responsibility for space activities is considered as a customary norm in international space law, somewhat of a higher level than the norms found in the treaties.[19]

The Liability Convention. The second relevant source of law regarding matters of liability is the Liability Convention. It is important to note that, as of the 1st of April 2019, the Outer Space Treaty has broader application than the Liability Convention because it has been ratified by 109 countries versus the 96 that ratified the Liability Convention.[20] The Liability Convention covers only the issue of liability so that it can be considered as lex specialis in relation to the Outer Space Treaty. However, States parties to both instruments might determine which one to invoke when they seek compensation. The relevant disposition for our purposes is Article III.

Note that Article II is not taken into consideration because it establishes a strict liability regime for damage caused by space objects on the surface of the Earth, for example, upon a failed re-entry causing damage to property before the start of outer space wherever that might be. The notion of strict liability entails that launching states are absolutely liable for damages caused by spacecraft even if they took all possible measures to avoid the event in which such damage was inflicted. Therefore, while better reliability engineering would reduce the risk of damage caused by failed launches and re-entries, it would not affect the liability regime of the launching States in such events. Therefore, it need not concern us, for the ongoing argument hinges on the presence of fault when operating a spacecraft for which sound

---

[18] Stephan Hobe, Bernhard Schmidt-Tedd, and K Schrogl. *Cologne commentary on space law.* vol. 1. Outer space treaty. 2009.

[19] R Venkata Rao, V Gopalakrishnan, and Kumar Abhijeet. *Recent Developments in Space Law: Opportunities Challenges.* Springer, 2017. isbn: 9811049262.

[20] *Status of International Agreements relating to activities in outer space as at 1 January 2021.* United Nations Office for Outer Space Affairs. A/AC.105/C.2/2021/CRP.10. http://www.unoosa.org/oosa/en/ourwork/spacelaw/treaties/status/index.html

software engineering practices are violated.

On the contrary to Article II, we observe that Article III does establish a fault-based liability regime for damage caused in outer space.[21] It dictates:

> "In the event of damage being caused elsewhere than on the surface of the Earth to a space object of one launching State or to persons or property on board such a space object by a space object of another launching State, the latter shall be liable only if the damage is due to its fault or the fault of persons for whom it is responsible."

Under international law, fault occurs if States fail to adhere to or breach an obligation imposed by law. Moreover, the presupposition of fault should be excluded if guidelines and standards have been complied with. Note that this is the case even if standards and guidelines are non-binding; the claim that software engineering practices are relevant for establishing fault is defended further in Section III. For now, it is important to stress that, in general, the body of space law is agnostic to size.[22] In other words, size does not matter concerning the liability of space objects, so that the same rules apply to space objects of varying sizes, ranging from big satellites as the forthcoming James Webb telescope to small satellites of only a few kilograms in mass.[23]

To sum up, liability for space activities – regardless of the size of the space object – is imposed upon States that (a) launch or procure the launch of a space object (b) launch a space object from their territory or their facility (c) if damage is caused by a space object to the property of another State or of persons, natural or juridical, of another State or to property of intergovernmental organizations. The aforementioned liability regime varies on the basis of where damages occur. On the one hand, if damages are caused on the surface of the Earth or to aircraft in flight States are absolutely liable, that is, strict liability is imposed for space activities that cause damages not in outer space. On the other hand, if damages are caused in outer space, the liability regime is based on fault. It is relevant to stress that States are

---

[21] Stephan Hobe et al. *Cologne Commentary on Space Law: Rescue Agreement, Liability Convention, Registration Convention, Moon Agreement*. 2013.

[22] Frans von der Dunk. *Liability for Damage Caused by Small Satellites—A Non-Issue?* In: Small Satellites: Regulatory Challenges and Chances. Ed. by Irmgard Marboe. Leiden: Brill, 2016.

[23] Jakhu and Pelton, *Small satellites and their regulation*; Irmgard Marboe. *Small Is Beautiful? Legal Challenges of Small Satellites*. In: Private Law, Public Law, Metalaw and Public Policy in Space: A Liber Amicorum in Honor of Ernst Fasan. Ed. by Patricia Margaret Sterns and Leslie I. Tennen. Springer, 2016, pp. 1–16. doi:10.1007/978-3-319-27087-6_1.; Nair, *Small Satellites and Sustainable Development: Solutions in International Space Law*.

responsible for the space activities carried out by their nationals. This contribution deals with the latter scenario and examines the effects of software engineering practices related to CubeSats.

C. On the relationship between fault, RTOS, and software certification

In the previous two subsections, we established that, firstly, in-orbit conjunctions are increasingly likely, and, secondly, that the conditions for a launching State being held liable for an in-orbit conjunction depends on fault. We now connect the notion of fault with spacecraft engineering practices, drawing attention in particular to the case of CubeSats and their most critical software components.

For space systems in general there is an ISO standard[24] stipulating safety requirements that makes recommendations concerning software, not only hardware. ISO 14620-1: Space systems — Safety requirements, highlights that "software that supports a safety critical function" should undergo a "formal software safety program consisting of software hazard analysis, software design requirements analysis, test, and verification and validation." The clauses of the standard most relevant to this study are 6.4.4.1 and 6.4.4.4, which respectively concerns software that implements or controls safety critical functions and software verification.

However, this work focuses specifically on CubeSats, rather than space systems in general. For small spacecraft, such as CubeSats, there is a dedicated recommendation,[25] which addresses the following problem:

> "Applying the same test requirements as those applied to traditional large/medium satellites, however, will nullify the low-cost and fast-delivery advantages possessed by small spacecraft."

Notably, the above-mentioned ISO standard for CubeSats makes allowances for the use of non-space-qualified commercial-off-the-shelf units (COTS) and does not explicitly cover software testing. Following this ISO standard, almost no software safety standards are set for CubeSats. There are however hardware testing requirements defined in the standard ensuring that non-space-qualified hardware is in fact adequate for purpose.

From the above evidence, we see that there is a significant gap between

[24] *Space systems — Safety requirements.* Tech. rep. 14620-1. ISO, 2018.

[25] *Space systems — Design qualification and acceptance tests of small spacecraft and units.* Tech. rep. ISO 19683:2017.

the standards for CubeSats and those for other spacecraft – the absence or presence, respectively, of the requirement to verify safety-critical software. For this reason, we focus on what we argue is one of the most critical pieces of software – the Real-Time Operating System (RTOS). An RTOS is a dedicated operating system that is installed on embedded systems in general, including, for example, aircraft, smart vehicles, IoT devices, or industrial appliances. They are particularly relevant for such applications, due to the requirement to precisely coordinate the timing of reading of various input sensors and response actions. Operating systems, in general, provide a layer that sits between the hardware and software of a system, exposing interfaces between components. For example, the RTOS enables a software application, which responds to telecommands for manoeuvring a satellite, to talk with the hardware that receives communications via antennae another piece of hardware that deploys thrust to implement a manoeuvre. Another example of a critical operation managed via the RTOS is battery management, ensuring that a payload, which typically consumes more power than is available, when fully operational, does not result in the failure of critical components. Notice that failure to properly manage either of these processes may result in the loss the telecommunications channel, possibly permanently. A telecommunications channel is a component indispensable for operational purposes, hence its failure turns the satellite into a dangerous object in orbit.

Indeed, a study spanning launches from 2009 to 2018 has indicated that almost half of satellite failures can be attributed to the failure of the communications system or power system,[26] both of which are managed via an RTOS.

While, failures of critical operations managing the communication channel may be caused by failures of hardware, for instance, the antennae, circuitry, or battery, causes due to failure of software should not be ruled out. Possible reasons for software to fail could be inadequate functional requirements, software bugs or even cyberattacks exploiting vulnerabilities.[27] In Figure 2, we provide a simplified fault tree which is a method safety engineers employ to communicate and measure causes of faults. By using fault trees and historic failure data for related hardware components and software, it may be possible for an engineer to measure the likelihood of the cause of a fault being due to a failure in the RTOS, relative to other types of

[26] Kara O'Donnell and Gregory Richardson. *Small Satellite Trending & Reliability 2009-2018*. In: Small Satellite Conference (2020). url: https://digitalcommons.usu.edu/smallsat/2020/all2020/185/

[27] P. J. Blount. *Satellites Are Just Things on the Internet of Things*. In: Air and Space Law 42 (2017). url: https://ssrn.com/abstract=3388549.

component failure. Note the hardware branch of the fault tree in Figure 2 can grow large as more detail is added by refining components into sub-components. Such a fault tree analysis is explicitly recommended to be conducted for spacecraft in ISO standard 14620-1,[28] hence adding a branch for critical software failures should not be overly demanding for engineers.

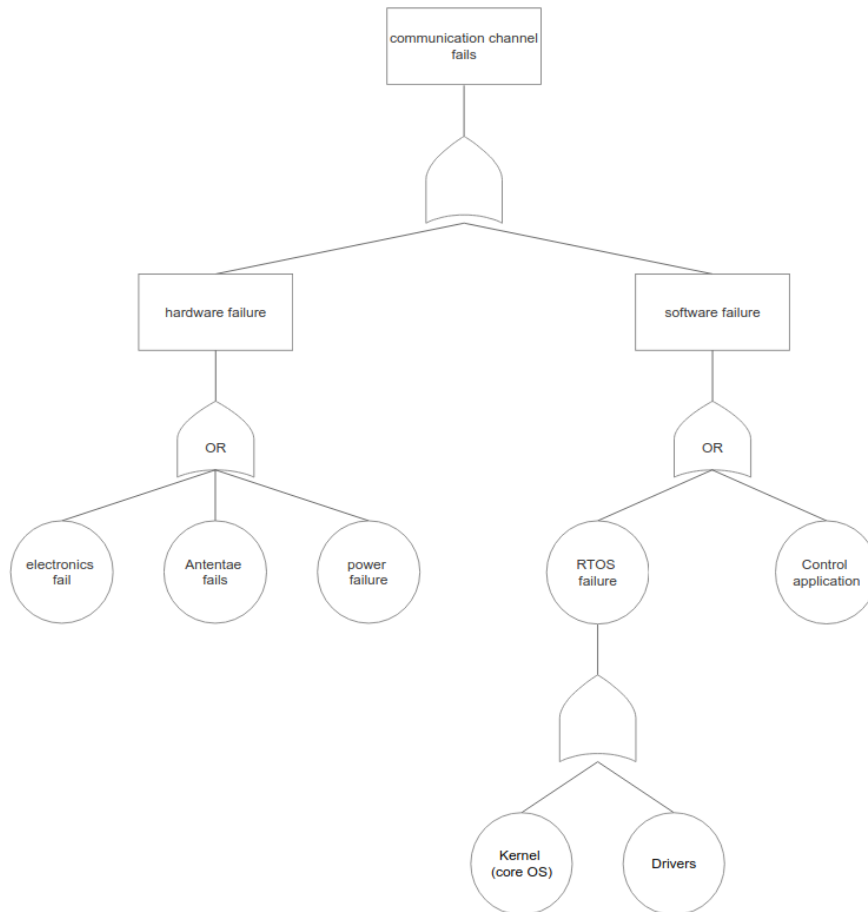

Figure 2: Sample of a fault tree suggesting possible causes for a telecommunications channel to be lost, indicating the RTOS Kernel.

The point of concern is that, for CubeSats, it is commonplace to make use of an RTOS which was designed by the open source community for non-space-going purpose, hence did not follow practices for software verification. For example, Amazon Free RTOS was designed for IoT sensor networks, for example, for gathering data in smart cities. Other RTOS projects used in CubeSats include eCos and Kubos. In the next section, we provide empirical evidence for our claim that some of these projects are likely unreliable even

[28] *Space systems — Safety requirements*. Tech. rep. 14620-1. ISO, 2018.

for CubeSat projects, whereas others appear to be better designed. Thus, even if obtaining an RTOS that meets the avionics standard for RTOS software (ARINC 653[29]) would unreasonably restrict the low-cost and fast-delivery advantages of CubeSat, it can be still reasonable to recommend that a RTOS with a good level of reliability is used in CubeSat projects. Similarly, demanding the highest level of software certification, such as EAL7 – a level of certification which almost no software has attained – would stifle innovation in orbit, by making outer space inaccessible to all but the largest multinationals and national agencies. Thus, it is important to measure what is a realistic level of certification to recommend, as we investigate in the next section.

While the failure of critical software may lead to channel failure, notice that if the software managing a payload, for instance, a camera and other remote sensing apparatus, fails, then it is easier to reinstate or even patch that software via the critical core of the CubeSat that manages the communication channel and its interface with the payload. Thus, requiring all software deployed on a CubeSat to meet high standards of dependability would impede innovation concerning the payload. For this reason, we restrict our focus to the RTOS. While it might also have been reasonable to take into account some critical applications that run inside an RTOS, it would not make sense to consider analysing the dependability of all software deployed on a CubeSat.

II. EMPIRICAL EVALUATION OF THE QUALITY OF CUBESAT REAL-TIME OPERATING SYSTEMS

This section concerns the technical evaluation of the quality of RTOS software deployed in CubeSats, which in previous sections we argued is the key possible cause of mission failure resulting in the creation of a dangerous object in orbit, which is not yet covered by CubeSat recommendations. Confirming whether or not a software project meets certification standards can take hundreds of person-years. For this reason, we adopt an empirical approach to back up substantiate our claim that improvements should be made in the quality of critical software typically deployed in CubeSats. In empirical software engineering we use metrics which are indicators of the

[29] P. J. Prisaznuk. *ARINC 653 role in Integrated Modular Avionics (IMA)*. in: 2008 IEEE/AIAA 27th Digital Avionics Systems Conference. 2008, 1.E.5–1–1.E.5–10. doi: 10.1109/DASC.2008.4702770; *Avionics application software standard interface part 3A: Conformity Test Specifications for ARINC 653 Required Services*. Tech. rep. ARINC Industry Activities, 2019, pp. 1–471.

coding style used in the project. Good coding style can indicate diligence in managing the complexity of a software project and hence can increase confidence that fewer vulnerabilities have been introduced via poor coding practice.

For this study, we considered four open source real-time operating systems (operating systems appropriate for control systems). Three – FreeRTOS, Kubos and eCos – are deployed in CubeSats. The fourth – seL4 – has been deployed on military-grade helicopters.[30] Note real-time operating systems meeting exceptionally high certification standards are deployed on aircraft (VxWare, LynxOS, Deos DO-178, INTEGRITY-178B, etc.). For example, the INTEGRITY-178B RTOS meets the second highest Evaluation Assurance Level set by the Common Criteria (EAL6[31]) – a software engineering standard. However, since, firstly, the source code of such RTOS projects is not in public domain, and, secondly, such software is likely out of budget for CubeSat operators, we take seL4 as our ground truth representing a verified RTOS. Interestingly, although seL4 is formally verified, it is not certified as being formally verified, since it used modern verification methods not yet acknowledged by the Common Criteria as being a replacement for traditional testing methods.[32]

In what follows we present some key metrics and suggest why the score for seL4 differs significantly from the other projects. We note that in each project we consider only C code, which forms the most of the code in all projects, except Kubos were 60% of the code is in the Rust language and 30% is in C. The Rust language has been designed to improve memory safety compared to C, which means that, for code written in Rust, the chances of software failures and vulnerabilities should be reduced. We used a tool by Spinellis et al.[33] to collect these metrics. In particular, we considered software complexity metrics and generic coding practices. We discuss them below.

---

[30] D. Cofer et al. *A Formal Approach to Constructing Secure Air Vehicle Software*. In: Computer 51.11 (2018), pp. 14–23.

[31] *Common Criteria for Information Technology Security Evaluation. Part 3: Security assurance components.* Tech. rep. 15408. Version 3.1, Revision 5. ISO, 2017. url: https://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R5.pdf.

[32] Gerwin Klein et al. *Formally Verified Software in the Real World*. In: Communications of the ACM61.10 (2018), pp. 68–77.doi:10.1145/3230627.

[33] Diomidis Spinellis, Panos Louridas, and Maria Kechagia. *The evolution of C programming practices: A study of the Unix operating system 1973-2015*. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE. 2016, pp. 748–759.

A. Size of the codebase

The size of the codebase (SLoC), that is the number of lines of code, is often used as a coarse-grained, yet relatively "cheap" metric that can serve as a proxy for reasoning about the complexity of a software project.[34] There are several studies in the software engineering community that explore the relation between the size of a code base and its maintainability, as well as software defect density.[35]

Consider first the seL4 project, with 45,781 SLoC. Verifying functional correctness of a project this size is still beyond the means of most companies. However, the most critical 8700 SLoC within the project have already been verified, which took around 18 person-years using current state-of-the-art verification technology.[36]

Now contrast the above to FreeRTOS, which features over 2.1 million SLoC. Assuming a linear relationship between SLoC and verification time (which is optimistic), verification effort for the entire project could exceed 4000 person-years. Even the effort for verifying a critical core of the system would likely be in the order of hundreds of person-years, which is beyond the budget of the biggest players in the space industry. Thus, SLoC is already a clear indicator that critical vulnerabilities are likely to be present in the FreeRTOS code base.

The eCos project also features a large code base just short of 1 million SLoC, which remains well outside the realm of verifiable software. The Kubos code base fares better with around 60 thousand SLoC, hence it may be possible to identify a critical core of the project to which verification techniques may be applied. Note that, although Kubos has not been formally verified like seL4, it does employ the Rust language which offers some lightweight guarantees.[37]

---

[34] Sheng Yu and Shijie Zhou. *A survey on metric of software complexity*. In: IEEE International Conference on Information Management and Engineering. IEEE. 2010, pp. 352–356.

[35] A. Gunes Koru et al. *An investigation into the functional form of the size-defect relationship for software modules*. In: Transactions on Software Engineering 35.2 (2009), pp. 293–304; Andrea Capiluppi. *Models for the evolution of OS projects*. In: Proceedings of International Conference on Software Maintenance. Los Alamitos, CA, USA, 2003, pp. 65–74.

[36] Gerwin Klein et al. *Formally verified software in the real world*. In: Commun. ACM 61.10 (2018), pp. 68–77. doi: 10.1145/3230627.

[37] Nicholas D. Matsakis and Felix S. Klock II. *The Rust Language*. In: Ada Lett. 34.3 (2014), pp. 103–104. doi:10.1145/2692956.2663188.

B. Halstead and cyclomatic complexity

There is a relationship between the code that has a high degree of complexity and the number of potential bugs/vulnerabilities.[38] The Halstead complexity metric calculates the data stream complexity and ignores control flow complexity (branches); it is language-independent and can be used as a predictor for the potential defect density. On the other hand, cyclomatic complexity captures control flow complexity, but ignores data flow complexity (thus, can be used to complement Halstead). See[39] for a comparison between Halstead and cyclomatic complexity measures and the "good" and "bad" values.

Complexity metrics can be a better indicator of the cost of certifying code than SLoC, since "System growth is not necessarily associated with structural complexity increases"[40] In other words, large software (SLoCs, files, and suchlike) does not have to be overly complex. However, these metrics do not have any absolute meaning – for instance, the fact that eCos has twice as high a Halstead complexity compared to seL4 does not necessarily mean that eCos is twice as expensive to verify as seL4. Therefore, we do not present the values on the y-axis of Figure 3, since the values are only approximate and relative indicators of complexity.

Cyclomatic complexity which considers the depth of structure of code, which can be used to indicate whether more expertise for an implementation to be correct. The mean cyclomatic complexity of seL4 is around 2.6, which is slightly lower than 3.2 for FreeRTOS and 5.2 for eCos, indicating it is generally easier to verify the control flow of code in seL4. A stronger difference can be seen by looking at files with the high cyclomatic complexity. If we look at files with high cyclomatic complexity above 12, seL4 features just one file with registering a cyclomatic complexity of 23, whereas FreeRTOS and eCos feature respectively 71 and 81 such files, peaking with a cyclomatic complexity of 131 and 194. Almost all complex

---

[38] Yonghee Shin and Laurie Williams. *An empirical model to predict security vulnerabilities using code complexity metrics*. In: Proceedings of International Symposium on Empirical Software Engineering and Measurement. 2008; Yonghee Shin et al. *Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities*. In: Transactions on Software Engineering 37.6 (2011), pp. 772–787.

[39] Yu and Zhou, *A survey on metric of software complexity*.

[40] Spinellis, Louridas, and Kechagia, *The evolution of C programming practices: A study of the Unix operating system 1973-2015*; Antonio Terceiro et al. *Understanding structural complexity evolution: A quantitative analysis*. In: 2012 16th European Conference on Software Maintenance and Reengineering. IEEE. 2012, pp. 85–94.

files in the FreeRTOS project are in libraries and vendor specific drivers, suggesting complexity is imported from 3rd-party components.

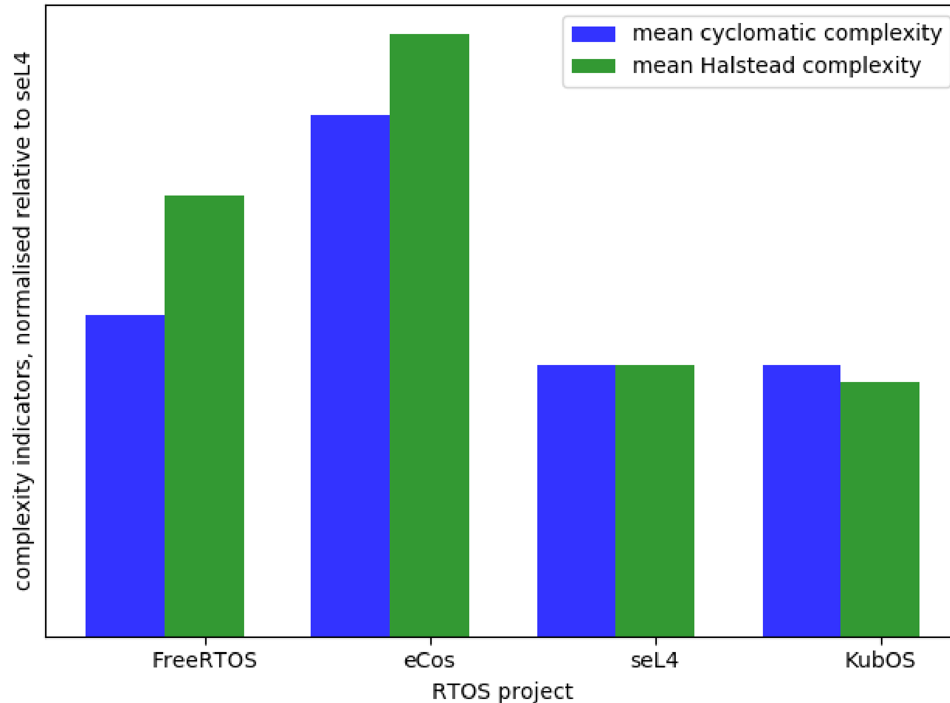

Figure 3: Normalised mean Halstead and cyclomatic complexity. Since these are relative indicators, the units on the y-axis are not shown.

The Halstead metric, which indicates the complexity of the data processed, independently confirms the above story. The mean Halstead complexity of seL4 is 270 whereas FreeRTOS and eCos register at 440 and 600 respectively. Also looking at the most complex files. As with cyclomatic complexity, the gap between FreeRTOS and seL4 widens when considering the files with highest Halstead complexity (for instance, the 92 most complex files in FreeRTOS, exceed the complexity of all but the most complex file in seL4). Having significantly more complex files may demand more advanced verification techniques, hence more human effort and expertise in order for the software to meet some level of certification.

The complexity scores for Kubos are comparable to seL4. However, recall, some of the more complex functionality may be written in the Rust language, and we considered only the C code in this evaluation.

C. Keywords: `goto`, `inline`, etc.

The extraneous use of the `goto` statement, a low-level keyword in the C language whose effect can almost always be captured by higher-level loops, can lead to the *spaghetti code* anti-pattern and significantly complicate program understanding[41]. This, in turn, leads to low maintainability of the code and a higher number of potential software defects.

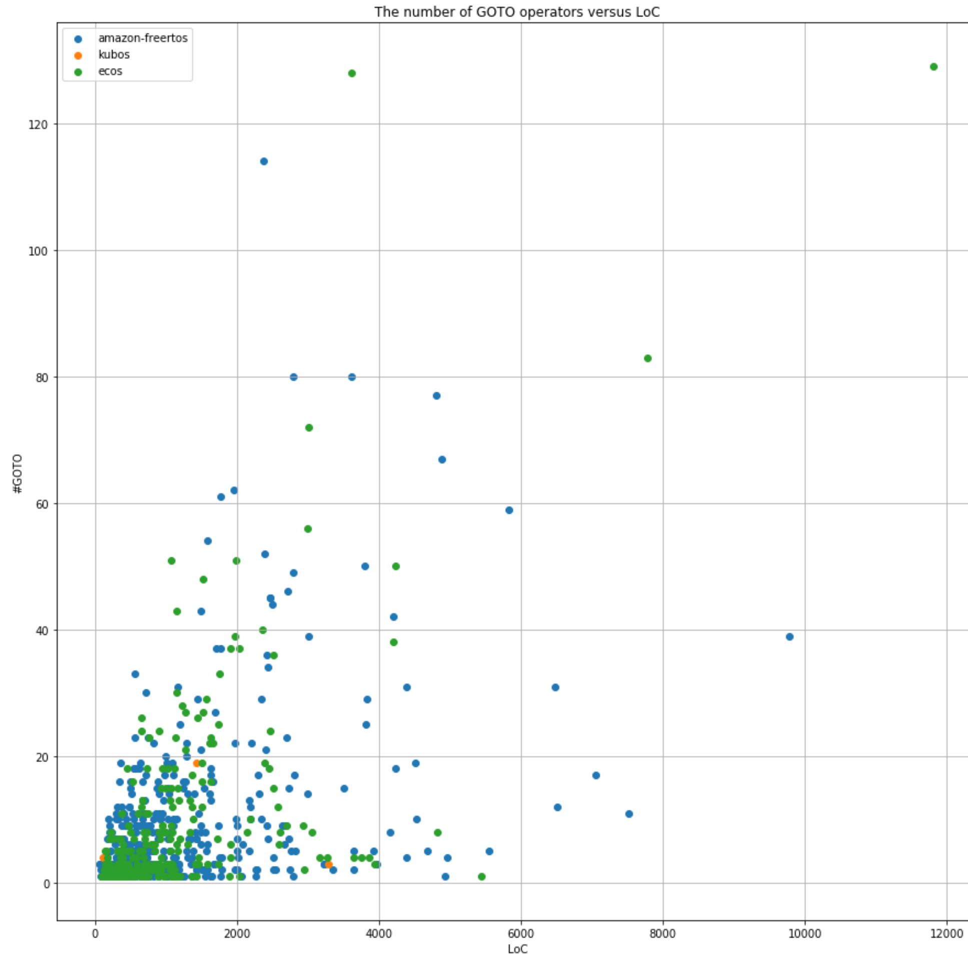

Figure 4: Scatter chart showing the density of `goto` *statements in files in the FreeRTOS, eCos and KubOS projects with at least one* `goto` *statement. In contrast, seL4 has zero* `goto` *statements.*

The project seL4 has zero `goto` statements, whereas the FreeRTOS and eCos projects contain 3922 and 2480 `goto` statements each. Since this is

[41] Edsger W Dijkstra. *Go to statement considered harmful*. In: Communications of the ACM 11.3 (1968), pp. 147–148.

amongst the clearest indicators separating seL4 from the other projects, we interpret our observation below for each project. A visualisation of the distribution of `goto` statements are illustrated in the scatter chart in Figure 4. This helps identify outliers where the number of `goto` statements are not linearly correlated with SLoC.

In FreeRTOS all these `goto` statements are in imported libraries and vendor specific drivers. The number of statements grows approximately linearly with the size of the file, with the notable exceptions being in a driver for the wifi vendor Espressif, which features the files with the most `goto` statements (between 67 and 114 statements). The above observations suggest that FreeRTOS may have vulnerabilities due to the libraries and drivers used, and, in particular, we could single out components that could be considered risky to use in a CubeSat running FreeRTOS, such as those developed by the wifi vendor Espressif.

The main outlier in the eCos project is the network packet implementation for IPv6 (ICMP), with 128 `goto` statements in a file of 3614 SLoc (indicated by the single isolated blue dot towards the top left of Figure 4). The eCos project consistently employs a high density of `goto` statements throughout the project, with a large proportion of files with at least 10 `goto` statement concerning networking. Since networking is an important function, this could be considered to be an alarming weakness of FreeRTOS worthy of further inspection before FreeRTOS is deployed in a CubeSat project where dependability is a requirement.

The 26 `goto` statements in Kubos are mainly contained in a JSON library. JSON is used for structuring messages and system configuration files, hence we recommend this library should be tested or verified.

Other keywords can indicate the quality of code. The presence of `inline` and `register` keywords in C code were introduced in earlier versions of the language to alleviate the deficiencies that the compilers had with allocating registers and inline functions. Since the compilers are getting better, extraneous use of such keywords should indicate that the codebase (or at least the part thereof) is quite old and has not been modified/refactored since a long time ago.[42]

---

[42] Spinellis, Louridas, and Kechagia, *The evolution of C programming practices: A study of the Unix operating system 1973-2015*; Gregory J Chaitin. *Register allocation & spilling via graph coloring*. In: ACM Sigplan Notices. Vol. 17. 6. ACM. 1982, pp. 98–105.

The significant presence of `inline` and `register` keywords suggests the presence of legacy code in FreeRTOS and eCos. While the seL4 project uses only 4 `register` keywords, FreeRTOS uses 175 and eCos uses 2113. The seL4 project uses 95 `inline` keywords, is still eclipsed by the 693 `inline` in the FreeRTOS project and 376 in the eCos project. Kubos fares better, since it contains zero legacy keywords.

III. LEGAL IMPLICATIONS FOR LAUNCHING STATES

The goal of this section is to defend the interpretation that a launching State could be liable if it authorizes CubeSats missions that do not meet certain software reliability requirements. In particular, we emphasise that the RTOS is a critical software component given its role in managing the software and hardware that manages critical functionality, notably the communication channel that enables a CubeSat from being controlled remotely. This would create an incentive for policy makers to introduce mitigation strategies at the national level, which would improve on the current state of the art.

Our task in this section is to ask if the conduct of licensing a CubeSat mission with a poor RTOS violates a duty established by law. Under current international space law, the answer appears positive. There are two interpretative paths to establish the fault of States that license a mission with an RTOS that does not meet specific software reliability standards.

The first interpretation is based on considering that Article VI of the Outer Space Treaty the norm that establishes the responsibility of States for activities in outer space – includes a specific standard of diligence which would be violated if States were to license space missions with inadequate critical software on-board. This argument is corroborated by Article IX of the Outer Space Treaty which establishes that States Parties shall conduct their space activities with due regard to the corresponding interests of all other space Parties. While the standard of conduct contained in Article IX is not precisely defined, it seems possible to concluded that "due regard" requires launching states to not authorize activities that pose a significant risk for the space operations of other States Parties (including their nationals). In this case, States would be held liable on the basis of international responsibility established by Articles VI, VII, and IX of the Outer Space Treaty.[43]

---

[43] James Crawford. *Articles on Responsibility of States for Internationally Wrongful Acts*. In: United Nations Audiovisual Library of International Law (2012). url: http://legal.un.org/avl/ha/rsiwa/rsiwa.htm

A second, but more tenuous, interpretative way relies on a creative interpretation of the Article IV of the Outer Space Treaty and may not be corroborated by expert opinion. In more detail, it could be possible to argue that licensing a CubeSat mission with a poor RTOS may constitute a breach to Article IV as it could hamper the peaceful use of outer space. This is because failures that may be caused by a faulty RTOS increase the risk of in-orbit conjunctions, and can contribute to environmental degradation, thereby preventing other parties from conducting affairs in space 'peaceably'.[44] The launching State should be held responsible if a CubeSat causes damage in outer space because of malfunctioning under Article III of the Liability Convention. It is important to stress that, in this case, fault is presupposed as a consequence of poor software engineering practices, so that the launching State would be able to exonerate itself if it proves that something else caused the damage. As, for example, if an anomaly in space weather caused it. Following this interpretation, the practical effect would be a reversal of the burden of proof. Simply put, it will be on the entity which caused the damage to prove that another event was the cause instead of its conduct. Conversely, it is usually the party who suffered the damage that has to prove that the conduct of the other party caused the damaging event.

Of course, due to the nature of this contribution, essential elements necessary to establish liability for space activities have not been discussed. For example, the issue of the kind of damages, the procedural aspects, as well as the empirical difficulty of reconstructing the event that caused the damage in outer space. Nonetheless, our aim was to show that the successful management of the new wave of small satellites, in the case of CubeSats, logically presupposes that these satellites run adequately dependable critical software. That is, CubeSats software ought to adhere to a set of standards to ensure that the risk of malfunction is mitigated. According to the interpretations offered above, the matter becomes urgent because launching States might have to bear the risks associated with poor critical software components. So that, alongside ethical and engineering reasons to impose requirements on RTOS there are legal ones as well. More importantly, we argued that an obligation to demand a certain degree of software dependability is already enshrined in international space law. On this basis, the next section examines three possibilities to address the issue at hand. Each solution will be discussed before arguing in favour of heightened requirements for the authorization of space missions.

---

[44] Kiran Krishan, Nair. *Small Satellites and Sustainable Development: Solutions in International Space Law.* Springer, 2019. isbn: 303018620.

IV. STRATEGIES FOR MITIGATING LIABILITY

This section explores the possible strategies to mitigate the liability of launching States for CubeSats missions without a dependable RTOS. It examines two options and concludes that it would be desirable for State to require more stringent controls on the quality of RTOS in the authorization process of space missions.

There are two possibilities to manage the risks of damages caused by CubeSats with non-dependable software on board. Balance is needed between robust certification procedures – for instance, Avionics standards, such as ARINC 653 – and a laissez-faire approach to software engineering practices. The peaceful use of space, along with environmental concerns, ought to be guarded by States in the licensing phase. On the one hand, strong software requirements are likely to require new procedures and extensive scrutiny by authorities on space missions, thereby leading to an increase in mission costs. On the other hand, the principle of freedom of access to outer space requires that licensing procedures do not excessively restrict the recent trend toward the 'democratization' of space. The problem here is of balancing values. In the previous pages, we have shown how some practices should be discouraged because they increase the concerns associated with the surge in CubeSats missions. Additionally, we argued that a laissez-faire attitude on software engineering practices could make launching States liable for damages caused in outer space. Therefore, a higher degree of scrutiny for RTOS is desirable, the issue then becomes how to achieve it. It seems clear that there is a possibility to over regulate in this instance. For example, avionics standards appear to be too much to ask in this case. This is because the decrease in the costs associated to the access to space is desirable and should be guarded. Against this backdrop, this section examines two strategies to address the issues at hand.

The first option is to intervene at the level of the authorization of space missions. The duty of States to authorize and supervise space missions is established by Article VI of the Outer Space Treaty. The Convention on Registration of Objects Launched into Outer Space (henceforth the Registration Convention) specifies the duty of registration on launching States, which presupposes an authorization process. Neither the Outer Space Treaty nor the Registration Convention indicate the requirements for the authorization of space missions, which, consequently, vary significantly. States could require the software deployed on CubeSats to meet specific requirements. We should note that States could require all space missions to

deploy highly dependable software, however, we focus our attention on CubeSats because other missions generally have a higher success rate. There are many options in this case, ranging from formal verification to testing. States should evaluate different solutions according to the objective of the mission under authorization. What is important here is that States could – to mitigate the risks explained above – intervene at the level of the authorization of space missions by mandating space operators to use critical software that meet certain criteria.

The second option does not require intervention on the authorization processes. Instead, it entails modifying or introducing specific insurance requirements for CubeSats missions. Many States require insurance for space missions. Therefore, a possibility is to introduce stronger insurance requirements for missions that do not provide guarantees concerning critical software components. A similar solution that considers unmanageable objects has already been discussed.[45] In this case, by mandating higher insurance requirements, an incentive to develop guarantees for the software would be created. Of course, this might have the negative effect of restricting access to space for missions with lower budgets. However, States could consider exceptions for some space missions such as the ones carried out for educational purposes. Yet, this might not be necessary as the software examined above – seL4 – is highly dependable and open source.

The first option appears more desirable. It would allow States to comply with the objectives and the spirit of international space law while protecting the space environment and alleviating the problem of space debris, which would increase with the number of uncontrollable space objects. Also, States could manipulate the degree of control by requiring different standards of certification. So that, for example, a private mission that deploys CubeSats with an open OS might be mandated to formally verify the software while educational missions – like the ones operated by universities – might only be required to perform software testing.

Regardless of the solution adopted, critical software components ought to be highly dependable. For these reasons, and because – as shown in Section III – poor software engineering practices might lead to the liability of launching States, it is desirable to implement measures to ensure the dependability of critical software systems deployed on CubeSats sooner rather than later.

---

[45] Ting Wang. *A Liability and Insurance Regime for Space Debris Mitigation*. In: Science Global Security 24.1 (2016), pp. 22–36. issn: 0892-9882.

V. CONCLUSION

CubeSats are here to stay, so is the current international legal framework regulating space activities. In light of the increase in the risks of operating spacecrafts due to the surge in launches of small satellites this contribution argues for the imposition of stricter requirements for critical software components of CubeSats in the authorization phase. We have put forward both legal and technical reasons to support this conclusion. On the one hand, it has been argued that current space law already contains a duty for launching States to demand certain standards for the software deployed on board spacecrafts. This duty arises from obligations already enshrined in the international legal framework governing space activities. Its relevance has been shown in the context of the liability for damages caused by malfunctioning CubeSats in the event of orbital-conjunctions. Simply put, if launching States aim to avoid the liability risk – we argue – they should impose stricter requirements for critical software components. On the other hand, this conclusion has been empirically supported by evaluating different RTOS deployed on CubeSats to show that better dependability is possible and in reach of space operators. More importantly, we have shown that better solutions are available without imposing excessive additional costs on CubeSats operators. On this basis, we provided two mitigation strategies. We concluded that including software requirements in the authorization of space mission at the national level is more desirable when compared to the other option. Namely, a mandatory insurance schema. It is only at the authorization phase that environmental and ethical concerns can be addressed effectively.