SPACE RADSIM: Binary-Agnostic Fault Injection to Evaluate Cosmic Radiation Impact on Exploit Mitigation Techniques in Space

Johannes Willbold[1], Tobias Cloosters[2], Simon Wörner[3], Felix Buchmann[3], Moritz Schloegel[3,4], Lucas Davi[2], Thorsten Holz[3]

[1]*Ruhr University Bochum, firstname.lastname@rub.de*
[2]*University of Duisburg-Essen, firstname.lastname@uni-due.de*
[3]*CISPA Helmholtz Center for Information Security, firstname.lastname@cispa.de*
[4]*Arizona State University, firstname.lastname@asu.edu*

***Abstract*—Over the past decade, the proliferation of Low Earth Orbit satellites, driven by lower launch costs, has revolutionized space applications, from communication to earth observation and weather forecasting. This trend also introduced a shift in hardware: Specialized radiation-resistant hardware was displaced by cheaper commercial off-the-shelf components. As a critical part of modern infrastructure, satellites attract cyber attacks and are subject to terrestrial and space-specific threats, necessitating effective security measures. However, cryptographic protections and exploit mitigations remain limited in productive satellite firmware. Academic research on satellite security only focuses on cryptographic protections, which raises the question if exploit mitigation strategies are suitable for satellites or impacted by space-specific factors, such as cosmic radiation.**

In this paper, we present the first systematic analysis of 381 small satellite designs, identifying the prevalence of commercial off-the-shelf hardware platforms in space projects and the availability of ready-to-use exploit mitigation strategies for satellite platforms. Since mitigations are seemingly available, we explore the effects of cosmic radiation on software-based exploit mitigations by implementing RADSIM, an automated tool for simulating single event errors (bitflips). Our study simulated over 21 billion faults in differently hardened satellite firmware binaries to assess the fault tolerance of exploit mitigation strategies in the presence of cosmic radiation. Our results reveal that some mitigations barely impact the fault tolerance, while others increase the error probability of hardened satellite firmware by up to 19%. These findings provide novel insights into the trade-offs between exploit mitigation effectiveness and radiation resilience, offering guidance to satellite developers on optimizing security in space-based systems.

1. Introduction

Over the past decade, satellites have become a vital part of our modern digital society. They provide critical services ranging from global navigation systems, communication, and weather forecasting to earth observations. In recent years, commercial and government agencies started pursuing space-borne solar energy and in-orbit manufacturing as part of the space age revolution. These novel developments have been made possible at such rapid speed due to cheaper access to space and reduced costs for satellite development using commercial off-the-shelf (COTS) components. These developments are commonly referred to as the *New Space Era* [37], which is often associated with smaller, cheaper, and, notably, *far* more numerous Low Earth Orbit satellites. By 2010, a total of 81 *small* satellites, i.e., satellites weighing less than 10kg, had been launched. Since then, this number has increased to 2,136 in 2023, and it is expected to increase by 450 by 2025 [38], which does not include Starlink satellites.

While this opens new economic and academic opportunities, it also makes space targets increasingly attractive to cyber attacks. Fundamentally, every satellite consists of radio communications equipment to be remote-controlled from a terrestrial Ground Station (GS) and an On-Board Computer (OBC) that manages incoming Telecommands (TCs) and responds with Telemetry (TM). Hence, satellites are remote-controlled, space-borne, and connected embedded systems. In rare but increasingly common cases, they may even be described as cyber-physical devices due to the increasing popularity and affordability of thruster technologies that enable satellites to control their orbit (at least to a certain degree). This, in turn, allows them to interact with their environment or to crash into other space assets if hijacked by malicious actors.

These factors lead to clear motives, especially for state-controlled actors. Indeed, in recent years, we have seen space assets and space-supported infrastructure become crucial during geopolitical conflicts. For example, providers like Starlink and ViaSat that facilitate remote communication have come under attack by Russia during its invasion of Ukraine [46], [72]. In total, recent research counted 124 cyber attack operations carried out in the context of this conflict alone [56]. This trend points towards a far more hostile space security environment in the coming years.

Given their exposed nature and crucial role within crit-

ical infrastructure, we would expect well-tested and secure systems that have undergone extensive research. Yet, we find a different picture in practice: While research on the security of space-supported terrestrial equipment, such as very small aperture terminal (VSAT) systems, has gained traction in recent years [54], [55], [74], research on the security of the actual satellites themselves is more elusive. In 2023, Willbold et al. were the first to publish a security analysis of satellites [76], demonstrating a severe lack of security measures. More specifically, their analysis shows (1) a lack of cryptographic protections to secure the initial access to satellites, and (2) a complete absence of exploit mitigation techniques to perpetuate defense in depth. In many cases, the satellite operators rely on security by obscurity rather than technical means.

Looking at existing academic research for cryptography in space applications, we find various papers proposing new space-oriented solutions [9], [13], [27], [36]. In contrast, we have not found equivalent papers regarding exploit mitigation techniques for space systems. This raises the question of whether technical challenges simply do not exist or whether they have not been studied so far.

When analyzing the difference between terrestrial and space-borne (embedded) systems, the most striking distinction is their environment. Space systems must deal with significant and frequent temperature differences, low gravity, extreme remoteness, and cosmic radiation. Thus, if there are technical challenges, they either reside in (one of) these factors or are a consequence thereof, such as unconventional computing hardware or isolated software ecosystems.

In the first part of this paper, we manually study 381 small satellite designs via open-source intelligence and attempt to identify the underlying hardware platforms used. We intend to see if there is a stark discrepancy between the computing hardware deployed in space and the one commonly considered in security research or in mainstream compilers that offer contemporary exploit mitigations such as stack cookies [22] or Control Flow Integrity (CFI) [1]. Our analysis reveals that the most common hardware platform is ARM Cortex-M (44.2%), and only 8.5% of designs use a radiation-hardened chip. Since ARM Cortex-M is a predominant platform for terrestrial applications [69], this rules out major technical challenges stemming from a particular hardware platform used, such that we analyze the problem from an operational perspective. With many hostile factors in space threatening modern electronics, the biggest impact is highly likely stemming from cosmic radiation. Cosmic radiation can induce single event errors known as bitflips, where a binary bit in the system's memory or processor unexpectedly changes from 0 to 1 or vice versa. This effect is caused by high-energy particles striking the charges representing the state in either memory or the processing unit. While cosmic radiation poses a challenge to electronic systems on Earth, the impact is significantly higher in space due to the lack of atmospheric shielding. In terrestrial systems, bitflips are relatively rare, but even here on Earth, they are so common in data centers that error correction code (ECC) memory is employed. Dependable systems research has long explored the consequences of bitflips on general software reliability [35], [47], [77], but their impact on exploit mitigation features remains unknown.

Based on the nature of many software-based exploit mitigations, we speculate they may increase the risk of errors when deployed in radiation-heavy environments. For example, stack canaries that enforce a specific canary value being matched, or CFI, where tags or whitelisted addresses are compared, provide ample surface where a single bitflip can disrupt the program execution. Yet, the impact of single event errors on exploit mitigations remains elusive: At some point, the risk of a malicious actor attacking a satellite will outweigh the risk of losing access to the satellite due to a bitflip in some exploit mitigation. The balance is further shifted by the increased numbers of satellites, increasing control capabilities paired with ever-increasing reliance on them, as well as continuous simplification of how to communicate with them. To allow satellite developers to make a more informed choice requires a precise measurement of the impact of radiation on different, widely available exploit mitigation techniques.

To this end, we systematically study available, production-ready, software-based exploitation mitigation techniques. We design and implement RADSIM, a tool capable of simulating single event errors by introducing bitflips into code and data. The first fully binary-agnostic and deterministic fault-injection approach requires no pre-defined static injection points or probabilities. In an extensive testing campaign, we exhaustively simulate all potential bitflips that may occur during the execution of two real-world satellites. In numbers, we perform *21 billion* fault injection experiments, more than *four* orders of magnitude more than in any prior attempt [15], [58]. Based on this vast empirical evidence, we derive the impact of radiation on exploit mitigations. We find that stack canaries increase the failure probability of our two satellite targets by 14% and 19%, while CFI increases it by 2.1% and 0.5%, respectively.

Contributions. In summary, we make the following key contributions in this paper:

- We conduct the first systematic hardware analysis of small Low Earth Orbit satellites through an extensive open-source intelligence survey, showing that radiation-hardened processors are rarely used in practice.
- Identifying radiation as a core threat to reliability, we propose RADSIM, the first fully binary-agnostic and deterministic fault injection approach, enabling us to evaluate the susceptibility of direly needed exploit hardening techniques to radiation.
- Performing 21 billion fault injection experiments, we are the first to study the impact of software-based exploit mitigations on fault tolerance and show that some techniques have barely any negative impact, while others increase failure probability by up to 19%.

We release the source code of RADSIM at https://github.com/CISPA-SysSec/space-radsim.

2. Survey of Applicable Mitigations

Low-level languages such as C/C++ require the programmer to manage the application's memory manually. As human errors are inevitable, errors in such parts of programs occur and lead to *memory corruption* bugs that enable attacks to overwrite data or hijack the program's control flow. Over the years, various mitigations have been proposed to hinder attackers from exploiting such vulnerabilities. Famous mitigations that found widespread adoption include stack canaries [22], shadow stacks [18], and Control Flow Integrity (CFI) [1], [17].

While these features are crucial to enforcing the security of systems, their real-world adoption usually relies on compilers supporting them. In the case of the aforementioned mitigations, modern compilers such as GCC and Clang have adopted them and allow users to enable them via compile-time flags. It is essential to consider that these features often rely on specific hardware and platform support; for example, GCC only supports shadow stack on x86 platforms due to the reliance on the Control Flow Enforcement Technology (CET) [28]. Similarly, while Clang ships with several features implemented in more abstract compiler pass layers, these features often require some form of Instruction Set Architecture (ISA)-specific changes. These ISA-specific changes are generally only implemented for popular platforms, such as x86 or AArch64; their support for niche platforms, such as those commonly found in embedded systems, is more elusive and depends on the specific case.

Hence, the chosen hardware platform determines which exploit mitigation features are available during satellite development. In the following, we first survey which processors and ISAs are commonly used for space systems. We then pick the most common platforms and survey the exploit mitigation features that the mainstream compilers GCC and Clang offer for these platforms. Finally, we select two open-source satellites that fit the typical hardware and platform criteria and apply the available exploit mitigations.

2.1. Systematization of Hardware

We surveyed all publicly tracked small satellite designs launched between 2020 and 2023 in Table 1. We did not include designs for 2024, since the primary source of technical details of these satellites is either research or engineering papers that take time to publish. Hence, recently launched satellites likely have far fewer numbers available to them. For a comprehensive list of satellites, we used the Nanosats Database that publicly tracks all satellite projects known to the public in some form [38]. The database has already been used in several research works and is generally considered the most comprehensive data collection on small satellite missions [39], [40].

For each satellite design in the 2020–2023 time frame, we manually investigated only *unique* designs. If a satellite design, say *Planet Labs's Dove*, has been launched over a hundred times, we only studied it once. This results in 381 unique satellite designs. We manually investigated them through open-source intelligence to determine the exact hardware platform, if publicly known. We could find data with sufficient confidence for 129 (34%) of the designs. In several cases, it is not explicitly stated that, for example, an STM32F4 is used, but it is noted that the bus system of a specific company is used. In these cases, we investigated this bus system, and if we could identify the hardware platform, we counted it accordingly. In rare cases, this may lead to misclassifications if the company uses an unannounced or one-off bus system. However, companies generally use flight-proven hardware instead of new developments, making this case negligible.

We aggregated the results, accumulated the hardware platforms per year, and noted them in Table 1. After the *Year* column, the first group of columns describes the most common hardware platforms. We can notice that with 57 instances (44.2% of 129), *ARM Cortex-M* is the most common hardware platform for satellite buses that have public information available. We grouped all designs in the left half with less than five occurrences under *Other*. The second group of columns only includes radiation-hardened processor designs, 11 designs in total. Notably, this means that only 8.5% of the satellite designs we could obtain data for include a radiation-hardened processor, making this the exception rather than the rule.

Another critical insight from Table 1 is that, most likely, almost all publicly known satellite designs rely on Real-Time Operating Systems (RTOS). Linux generally requires a Memory-Management Unit (MMU) to manage virtual memory and user processes. While it can be compiled without MMU, it is usually considered a niche application. This also matches the results of prior work's satellite case studies [76].

Based on our survey, we assume that *ARM Cortex-M* is the most widely used hardware platform in practice. However, the lack of publicly available data for 66% of the satellite designs leaves room for some uncertainty. At the same time, our dataset suggests that radiation-hardened designs are relatively uncommon. We speculate that such designs are more likely to be used in closed-source scenarios, such as industry or military applications, where the higher cost of radiation hardening is justified.

2.2. Availability of Exploit Mitigations

Our previous survey identified 15 different hardware platforms, but only three have ten or more occurrences. We pick only these to further investigate their compiler-level support for exploit mitigations. Table 2 shows the availability in GCC and Clang of the most popular exploit mitigation features for these platforms.

We focus on the defenses implemented in upstream compilers and do not include research prototypes. While research papers may implement their prototypes on top of LLVM, their prototype nature makes them generally unsuitable for production environments, especially considering the high-reliability requirements of space systems. Further, we consider it highly unlikely that space systems engineers would adopt research prototypes rather than widely

TABLE 1: Overview of CPUs used in satellite on-board computers (OBCs), from 2020 to 2023, based on open source intelligence gathering. Note that results for closed-source satellite OBCs may vary. We group conventional CPUs with less than five occurrences under *Other*; *ARM-C* is short for *ARM Cortex*.

	Conventional CPUs							Radiation-hardened CPUs				No data
Year	**ARM-C M**	**ARM-C A**	**AVR32**	**ARM9**	**AVR8**	**Other**	$\sum$	**MSP430**	**LEON3**	**PicoSkyFT**	$\sum$	
2020	10	7	2	4	2	-	25 (51%)	1	-	1	2 (4%)	22 (45%)
2021	8	1	2	3	3	4	21 (42%)	1	-	-	1 (2%)	28 (56%)
2022	18	5	4	1	1	10	39 (30%)	4	4	-	8 (6%)	85 (64%)
2023	21	2	2	1	1	6	33 (22%)	-	-	-	0 (0%)	117 (78%)
$\sum$	**57**	**15**	**10**	**9**	**7**	**20**	**118 (31%)**	**6**	**4**	**1**	**11 (3%)**	**252 (66%)**

available, easy-to-access compiler features. Several papers track the availability of exploit mitigation features across hardware platforms [2], [69], [71] and operating systems, focusing on academic prototypes to track existing research; however, we want to display the current state of real-world production-ready availability. In the following, we briefly introduce each defense.

Stack Canaries. Stack cookies or canaries [22] are guard values placed on the stack between local data and the return address of a function. Before returning from the function, the canary's value is checked against a global ground truth, thus detecting any modification that inevitably occurs when attackers attempt to overflow a buffer to overwrite the return address with an attacker-controlled target address. Both GCC and Clang support stack canaries in different flavors: Either all functions can be protected or only functions that use buffers of a specific size or that call `alloca`.

SafeStack. A SafeStack, part of the code-pointer integrity project [41], separates the stack into an unsafe and a safe area. The latter stores return addresses and other data that cannot be overwritten by an attacker, such as register spills. Thus, buffer overflows on the unsafe stack cannot overwrite control data. This feature is only available in Clang.

Shadow Stack. Like the SafeStack, a shadow stack [18] protects control data by placing return addresses on a second stack. In contrast to the SafeStack, the shadow stack only stores return addresses instead of mirroring the full stack. Shadow stacks are available for both GCC and Clang.

Control Flow Integrity (CFI). CFI [1] ensures that only valid edges are taken. Previously outlined mechanisms focused on protecting *backward edges*, i.e., returning from function calls to the caller. For CFI, we focus on *forward edges*, i.e., virtual and indirect function calls. Before each call, compiler-inserted code ensures the target is valid, using allow lists or tags. In the remainder, we refer to the allow list version, which employs compile-time static type checking as CFI, and to the tag version, which checks the function types dynamically at runtime, as function sanitization.

GCC implements CFI only for C++ *vtables* using Vtable Verification and provides support for Function Control Flow protection (`-fcf-protection`). Still, that feature requires Intel's CET, which makes it accessible only to x86 [70]. Shadow stacks (`-mshstk`) require the same technology [28] and *SafeStacks* are not implemented. This makes GCC unsuitable for hardening embedded devices firmware, especially in our case.

LLVM supports CFI in the allow-list and tag flavor described above on ARM Cortex-A and M as well as stack canaries, which we tested. While LLVM generally also supports SafeStacks and shadow stacks, they lack support for the bare-metal API triple (*arm-none-eabi*). When enabled, upon function start, LLVM calls a special function that needs to be developer-supplied to set up the SafeStacks, which essentially amounts to missing support.

While GCC and LLVM readily support ARM Cortex-A and M, AVR32 differs. All occurrences of AVR32—not to be confused with AVR 8-bit commonly deployed on Arduinos—stem from the satellite integrator *GomSpace*, which deploys the chip in its *NanoMind A3200* board. Only a GCC toolchain delivered by Microchip Technology Inc. appears to implement AVR32 compiler support [7]. We could not test the support of stack canaries for this target. However, considering the overall unsuitability of GCC for our case, we disregarded it as an option (cf. Section 6).

2.3. Target Selection

We have found that ARM Cortex-M is the most proliferated satellite platform, and only LLVM supports relevant exploit mitigation techniques. We select two targets to evaluate the impact of exploit mitigations on satellites.

2.3.1. ORESAT. ORESAT 0.5 is an open-source 2U CubeSat, the most common form factor for small satellites [40]. The satellite launched on August 16, 2024, as part of the OreSat initiative to promote accessible space technology and education. The satellite's software is developed in C and runs on an STM32F439 microcontroller, which serves as the main Command, Communications, and Control (C3) system. This microcontroller manages critical functions and facilitates communication between various subsystems, including the Ground Station, using a custom Telecommand (TC) and Telemetry (TM) format for Ground Station communication and the standardized ECSS CANbus Extension Protocol for internal satellite communications. ORESAT employs the RTOS ChibiOS RTOS [30], [51], [66].

2.3.2. ACUBESAT. ACUBESAT is an upcoming open-source 3U CubeSat aimed at advancing accessible space

TABLE 2: Overview of compiler-level support for exploit hardening techniques [28], [70], [71] for the three most common CPU platforms in satellite OBCs.

Platform	Compiler	Stack Canary	Safe Stack	Shadow Stack	CFI	FSan
ARM Cortex-M	GCC	✓	✗	✗	✗	✗
ARM Cortex-M	LLVM	✓	✗	✗	✓	✓
ARM Cortex-A	GCC	✓	✗	✗	✗	✗
ARM Cortex-A	LLVM	✓	✓	✗	✓	✓
AVR32	GCC	?	✗	✗	✗	✗
AVR32	LLVM	–	–	–	–	–

technology and providing educational opportunities. The satellite's software is developed in C++ and operates on an ATSAMV71Q21B microcontroller—a 32-bit ARM Cortex-M7—and it employs FreeRTOS, a real-time operating system. Telecommand and telemetry communications adhere to the ECSS-E-ST-70-41C standard, a widely recognized protocol in the aerospace industry [3], [16].

In conclusion, we have seen that the ecosystem mainly evolves around the same platforms commonly deployed in terrestrial applications and is widely investigated in academic research. We have also identified ACUBESAT and ORESAT as representative samples in our further investigation. In what follows, we will explore the impact of space-based radiation on exploit mitigations in satellite firmware. Since exploit defenses often rely on comparing certain (random) values, like stack canaries or function pointers and tags in CFI, it can be reasonably assumed that bitflips over-proportionally impact them. Therefore, it seems plausible that such measures are not deployed due to uncertainty around their impact in high-radiation environments.

3. Single Event Errors

Single Event Errors (SEEs) are unintended disruptions in a system's memory or processing components caused by external factors, particularly ionizing radiation from cosmic rays or radioactive decay. These disruptions can lead to bitflips in digital circuits without any permanent damage to the hardware. SEEs have traditionally been a concern in high-reliability environments, such as aerospace and safety-critical systems [32].

The fundamental mechanism behind SEEs is the impact of high-energy particles, which can cause localized ionization within microelectronic devices. This ionization induces transient electrical charges, changing stored data or logic states in memory cells or processor registers. Due to greater cosmic radiation exposure, such bitflips are notably more common in space or high-altitude environments. They can also occur at ground level, where lower energy particles can still affect advanced microelectronics. Transient memory faults caused by SEEs are often modeled as random single-bit errors due to their typical sparsity and distribution [23].

The frequency of such faults appearing depends on the current solar activity and the level of protection the space assets receive from external factors such as Earth's Van Allen belt that shields radiation, or from radiation hardening measures deployed on the spacecraft. The specific number of bitflips similarly varies; for example, a recent satellite measured 2,128 Single Event Upsets (SEUs) over a period of 286 days, statistically amounting to one SEU every 3.4h [49]. Such transient memory faults caused by SEEs are especially concerning when they result in Silent Data Corruption (SDC), where data errors occur without immediate detection.

3.1. Modeling SEEs

We rely on existing literature, primarily from the dependable systems research area, to accurately model SEEs [15]. Since the exploit mitigations we evaluate rely on values stored in RAM and code segments, we focus on the physical storage media for this memory, primarily Static-Random-access Memory (SRAM). Prior work has shown that the probability of particle strikes is independent of the previous particle strike [43]. This assumption allows us to observe bitflips independently without considering that the probability distribution for the next bitflip differs conditionally. Further, SRAM faults are uniformly distributed; hence, each bitflip in an SRAM region is equally likely as in any other region [67]. These assumptions allow us to view each bitflip independently, as the probability of *when* (temporal component) and *where* (spatial component) are independent.

Next, we must consider how many bitflips we must consider at each step. Previous research has shown that multi-bitflips, where a single particle causes multiple bits to flip, can be caused by a particle hitting a *memory cell* that, for example, holds a 4-bit state, causing it to jump multiple levels. Further, depending on the type of radiation and the memory designs, a single particle strike may also cause an effect on multiple cells [25]. Intuitively, multi-bitflips have a higher chance of outright crashing a system compared to single bitflips. Since we want to evaluate the impact of exploit mitigation techniques on crash rates, it makes sense to look into single bitflips, as those have the lowest chance of crashing a system on their own. If the system crashes due to bitflips anyway without exploit mitigations, it will also crash with them.

Additionally, Sangchoolie et al. have shown that single bitflips have the highest chance of causing SDC, i.e., bits that impact the program's outcome without crashing it [58]. From an exploit mitigation perspective, such SDCs might be turned into actual crashes, for example, when corrupting a code pointer, which CFI would detect.

In conclusion, we want to consider single bitflips, which are independent in their temporal and spatial dimensions for memory regions.

4. RADSIM Design Challenges

In this section, we elaborate on the challenges involved in evaluating the impact of binary hardening on full system firmware. We must account for the impact on non-hardened binaries as a baseline to observe the impact. We discuss the

methods of related work and whether these strategies are effective in this regard. Finally, we propose the high-level concept for RADSIM.

4.1. Problem Statement

Our goal is to answer the open question of whether radiation-induced bitflips impact exploit mitigation features deployed in satellite firmware. So far, Table 1 established that ARM Cortex-M is the most commonly deployed CPU. Further, these satellites use bare-metal firmware, i.e., they contain a compiled-in RTOS and do not run in the userspace of an OS like Linux. This already raises the first challenge. We need a full system emulation approach to run the full system firmware for our fault injection. Consequently, since bare-metal devices do not have the same notion of input and output via defined standard interfaces, we need to establish a method of using MMIO read as input to the program and MMIO writes as output from the program to check whether it successfully performed its operations.

We also want pure software fault injection since we are evaluating pure software mitigation features. We do not need extensive fault injection on the simulated transistor level or cycle-accurate processor simulation. Such simulations have recently been shown by Papadimitriou et al. [52], who presented the significance of identifying the exact location in the full stack of hardware and software where faults manifest. For our purpose, elaborate hardware simulations include unnecessary noise and performance overhead unrelated to binary hardening strategies.

Furthermore, we want to compare the exact executions of the same input between two programs, which requires *deterministic behavior*. This way, we can compare exact execution traces of a hardened and non-hardened binary and localize faults to see if the hardening technique manifests in any significant form. Otherwise, comparing executions of different inputs will cause significant noise due to, e.g., functions with stack canaries being called more or less often than by the input of the other binary.

Finally, from the previous replayability requirement, we can deduce that any executions in the fault injection approach must also be fully *binary-agnostic*. In particular, when and where to inject a fault must not depend on the binary under test because a stack canary-enhanced software executes more instructions for the same input. Therefore, the fault injection must not depend on the current program counter (PC), the number of basic blocks, or similar factors.

4.2. Related Work

Existing research on fault injection, especially from the area of dependable systems, is plentiful [19], [29], [44], [53], [73]. In general, fault injection approaches can be divided into approaches that use precise simulators, such as gem5 [14], or an emulator such as QEMU. As per our requirement, we do not need sophisticated simulations. Thus, we focus on emulator-based approaches. In the following, we consider several approaches relying on QEMU as an emulator framework for a closer comparison.

QEFI [20] uses either a probability-based trigger or a GDB-based interrupt trigger, such as the PC, to decide when to inject a fault. As discussed above, a PC-based or similar trigger does not work for our approach, nor does a probability-based trigger. Similarly, An et al. use a GDB server to inject faults based on a pre-defined trigger [6], as well as Sini et al., who propose a QEMU testbench setup [65]. Holler et al. [34] present work on fault modeling, i.e., which type of fault to use, while also using pre-defined fault triggers with static addresses. Ferraretto et al. present an approach that can treat different fault types but also relies on a probability of whether a fault is injected, making it equally unsuitable [26]. Further, it assumes Linux userspace applications, breaking our bare-metal requirement. Hauschild et al. present a plugin for the QEMU-internal Just-in-time code compiler (TCG), which allows for platform-independent fault injection [33]. Besides several different parameters, the work also uses a fault address to describe where a fault should appear. Recently, Almeida et al. [5] used QEMU-internal modifications to inject faults at different places, such as instruction translation or memory accesses. However, the approach also relies on pre-defined addresses to trigger the injection.

In summary, previous work either relies on pre-defined fault injection points via PCs or uses a fixed failure probability for faults to occur at specific times. However, both strategies are not binary-agnostic: PCs change when recompiling a binary with different exploit hardening techniques due to the additional code. Hence, the PC-based method does not allow for a meaningful comparison without extensive (and often manual) PC alignment efforts. Similarly, timing- and probability-based fault injection approaches are also not agnostic, as execution timings differ due to the code changes. Moreover, probability-based methods carry the problem that the values we are interested in have a small size (i.e., a canary consists of only four bytes). Hence, we assume that any noise introduced due to the failure probability will likely overshadow any measured effect. Due to these limitations, previously proposed approaches are unsuitable. Instead, we need a a novel method that captures all possible impacts on the security techniques in a binary-agonistic way.

4.3. RADSIM Design Concept

Intuitively, limiting the value domain appears unavoidable. A fault must be injected at selected places or due to a probability. To this end, to be fully binary-agnostic and deterministic, we plan to evaluate every possible bitflip, even if it appears unfeasible at first glance. Bitflips can occur in any memory cell (spatial dimension) and at any point during program execution (temporal dimension). Attempting to exhaustively simulate bitflips across both dimensions would result in a combinatorial explosion, rendering comprehensive analysis computationally infeasible.

To overcome this challenge, RADSIM utilizes an approach that effectively reduces the two-dimensional prob-

lem of spatial and temporal bitflip occurrences into a one-dimensional analysis. The core principle lies in focusing on memory read operations as critical points where bitflips manifest observable effects on program behavior. By iterating through all memory read operations and injecting bitflips into every possible bit of the data being read, we capture the impact of bitflips precisely when they influence the execution flow. The rationale behind targeting memory read operations is based on the observation that a bitflip in memory affects the program only when the corrupted data is accessed and utilized. By aligning bitflip injections with memory reads, we inherently account for the temporal aspect of when a bitflip would affect the program. This alignment allows us to simulate the effect of bitflips occurring at the exact moment they would have the most significant impact, such as during the checking of stack canaries—and a canary only exists for a brief period—or CFI validations. Furthermore, we only target read operations and not write operations because only a value being read into a program can have an effect, and any bitflip introduced by a write operation has to be read (where we inject bitflips) before having an impact.

By systematically flipping each bit in read data at each memory access, we exhaustively cover all possible single-bit corruption scenarios without limiting or explicitly iterating over the time dimension. Hence, in the following, we will describe an approach to efficiently evaluate all single-bitflip scenarios for a target program given an input.

5. RADSIM Implementation

We now describe our prototype implementation of RADSIM consisting of four components depicted in Figure 1. ❶ We use the state-of-the-art multi-input stream firmware fuzzer HOEDUR [61] to generate test cases via fuzz testing the plain variant of the target. ❷ We replay the inputs on the plain flavor to extract an MMIO reads and writes reference trace against which we can compare the plain variant later. ❸ In the next step, we replay *plain* MMIO reads on the hardened flavors, e.g., the stack canary binary, and check that the execution flow matches. We re-record the execution and create the necessary reference traces for each flavor. ❹ Finally, we run the reference traces for each flavor in our RADSIM bitflip engine to successfully test every possible bitflip in the firmware at any point during the execution.

Our implementation of RADSIM is based on HOEDUR but has little in common with typical fuzzing. Instead of focusing on fuzzing techniques, RADSIM utilizes the firmware execution environment that allows us to re-execute previous runs using the aforementioned traces to load snapshots, modify the memory for fault injection, and more. While we use HOEDUR to generate test case data (as shown in step ❶), this could be replaced by other input generation approaches, such as a software-in-the-loop simulation or similar techniques. In total, RADSIM is implemented in roughly 4,000 lines of new code.

5.1. Generating & Replaying Test Cases

Our RADSIM approach relies on dynamic testing, where we execute the target firmware to test the impact of bitflips. We need inputs that satisfy input constraints to execute the target firmware to reach the code commonly executed on the target in an operational environment. Unit tests and similar tests supplied through the firmware development process are usually insufficient, as they commonly focus on isolated aspects, such as a specific parsing functionality. Hence, we rely on full system fuzzing to automatically create full system test cases, where the entire firmware is rehosted and executed. The inputs generated during this fuzzing process can then be replayed, yielding *test cases*. The purpose of these *test cases* is to include all I/O data that would make runs between different executions and flavors indeterministic. This I/O data consists of interrupts and MMIO operations, which are recorded and stored in each *test case*. Another common source for I/O is Direct-Memory Access (DMA), which HOEDUR does not support directly, but only by treating it as MMIO accesses. Therefore, our approach implicitly includes rudimentary DMA support. This way, our test cases capture all potential I/O and can replay them between executions, which eliminates all I/O-based randomness and indeterminism. In summary, using this approach, we only have a single source of I/O (and therefore randomness), which eliminates the indeterminism. We record this I/O once on the plain flavor and then replay it across the hardened flavors.

Further, our approach aims at comparing differently *flavored* binaries, i.e., comparing a non-hardened (*plain*) variant with a stack canary variant. To properly compare runs between these similar but ultimately different binaries, we need to be able to *replay* an input from the *plain flavor* on any *hardened flavor*.

Finally, since fuzzing campaigns generally generate more test cases than our RADSIM approach due to limited computing resources, we must pick a subset of inputs from the entire set. For reference, in our evaluation, we found that exhaustively testing a single input takes 75 minutes using 98 cores, resulting in 122.5 core hours, making it unfeasible to test thousands of inputs.

In the following, we describe our fuzzing setup, our setup to replay test cases across flavored variants, and elaborate on our test case selection strategy.

5.1.1. Initial Fuzzing. Our work utilizes HOEDUR, a specialized firmware fuzzer designed to enhance firmware fuzzing effectiveness through multi-stream input handling [61]. Traditional firmware fuzzing methods typically interpret inputs in a flat, sequential manner, often leading to instability due to the mixed handling of data from different hardware interactions. HOEDUR addresses this by introducing a multi-stream approach that separates input streams based on distinct MMIO addresses. This distinction allows for precise targeting and manipulation of data streams, reducing the risk of unintended cross-stream interactions. Additionally, HOEDUR integrates MMIO modeling from

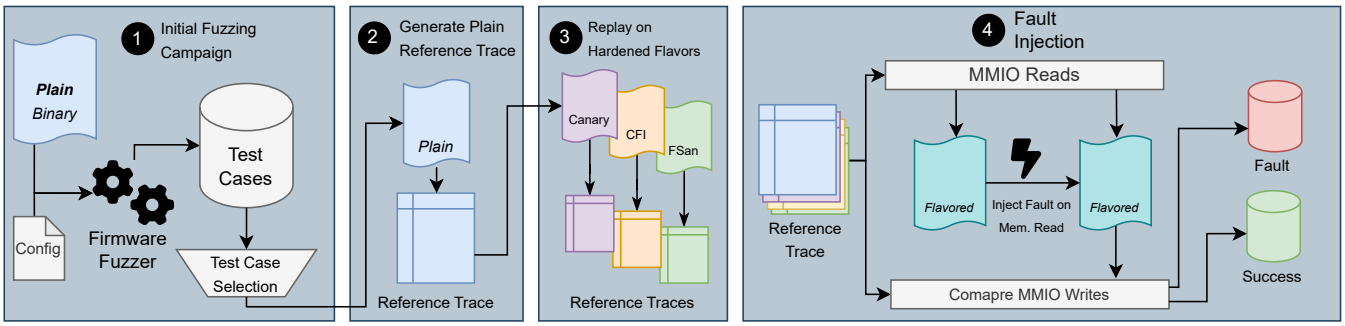


Figure 1: Overview of RADSIM

Fuzzware [59], which dynamically reduces the input space for each MMIO access, streamlining fuzzing efforts and enhancing stability and reproducibility in fuzzing sessions. Internally, Fuzzware uses symbolic execution to predict if, e.g., a four-byte MMIO access is being fully utilized. In practice, only a single bit might be used to determine a flag out of the four accessed bytes. Hence, the fuzzer only has to account for the single bit, drastically reducing the input space and increasing efficiency.

By default, HOEDUR identifies the input stream via a pair of PC and MMIO addresses, i.e., if the same MMIO address is read from two different PC addresses, it creates two separate input streams. Since PC addresses differ between our flavors, we opted for the input stream identification that only utilizes MMIO addresses; while this might degrade fuzzing performance, it is necessary to ensure replayability.

We fuzzed each plain target flavor until we achieved sufficient coverage in areas of interest. In satellites, the main focus area is the telecommand handlers and protocol decoding stack since they are the main code path parsing potentially corrupted data, which has also been the main focus area of previous research [31], [60], [75]. We refer to the full set of fuzzing inputs generated during the initial fuzzing as *test cases*.

5.1.2. Flavored Binary Replay. In the next step, we use the *test cases* and replay them in the hardened binary flavors. Initially, we intended to replay the inputs using the more solid multi-input stream approach from HOEDUR, which would tolerate a certain degree of out-of-order replay regarding MMIO inputs. However, we noticed that the Fuzzware MMIO models that HOEDUR relies on for each MMIO read differ between the flavors. This likely stems from the symbolic execution used to calculate these access models having issues with stack-canary functions.

To overcome this, we first replay the plain variant and record the exact four-byte MMIO values delivered to the firmware and their precise order in an *MMIO trace*. This way, the result of the MMIO modeling is already baked into the four-byte value. In the next step, we run the flavored binary and use the MMIO trace from earlier. This way, we can replay the input without needing the MMIO models.

Further, for the flavored binaries to correctly replay, they must be fully deterministic. They must not utilize any counter tied to their execution time, such as the number of executed basic blocks or the PC; this stems from the binary-agnostic requirement we established earlier (cf. Section 4.1). Several RTOSs rely on a `systick` feature provided by the processor, which emits periodic interrupts. Since this feature is tied to execution time, its exact timing would vary between variants. Hence, targets must be adapted not to utilize this feature, or the `systick` feature must be triggered only at pre-defined points that do not vary between variants.

Other sources of indeterminism can stem from interrupts, and general I/O. Interrupts in our setup are only emitted by the fuzzer, allowing us to take control of them. Hence, interrupts are baked into the replay traces, which makes them same across

After properly replaying the test cases on the defended variants, we record full RAM and MMIO traces, which detail all read and write operations. We later use these traces to determine whether the firmware is still behaving as expected during our fault injection and to determine the order and place of RAM reads.

5.1.3. Test Case Selection. The previously described fuzzing process produces a significant number of test cases, each yielding a unique set of edges in the target binary. Evaluating all test cases with our approach is not feasible; hence, we pick a subset. We opt to cover as many parts of the program as possible, and our subset should maximize code coverage. Therefore, we calculate the *block coverage* for each test case using a bitmap akin to the original fuzzing process. In detail, we transform the blocks covered by each input into a bitmap using a bloom filter. This yields bitmaps in which a higher number of set bits indicates a higher coverage, and equal bits in different inputs come from the same covered basic block.

We first select the input with the highest bitmap coverage and then, from all the remaining inputs, we select the one with the most additional bits set. We continue this until we have covered the entire bitmap. In simple terms, the selected test cases represent a minimized fuzzing corpus that covers every basic block (however, not necessarily every path).

5.2. RADSIM Engine

We develop a custom radiation simulation engine based on HOEDUR, internally based on QEMU, a broadly used emulation engine. Beyond the previously described test case generation, we also leverage HOEDUR's ability to manage and manipulate a firmware's execution flow, particularly the snapshot feature [61]. On a high level, we load the target firmware and the target test cases into our prototype, which yields one execution of the target. Then, we step through the program to *inject our bitflips* (cf. Section 5.2.1). Halting at every basic block, we iterate through each memory access (i.e., RAM and `.text` access). For each memory access, we iterate through each accessed *bit*, and for each bit, we flip that and only that bit and continue the program's execution without further stopping. During that execution, we *track the execution* (cf. Section 5.2.2) by comparing it to the reference trace, e.g., we check if the execution flow or the MMIO reads or writes change, and we record that accordingly. After the execution is complete, we restore the snapshot recorded at the basic block where the introduced bitflip is and move to the next bitflip.

In the following, we describe the bitflip injection part and the execution tracking in more detail.

5.2.1. Bitflip Injections. We load the target test case and firmware into our corpus. First, we process the data from the test case to extract a basic block execution trace that lists the memory accesses performed at each basic block. Notably, this is not done per unique basic block but per basic block in execution order. We start the program's execution and keep it running without interfering until we arrive at the defined starting point. In our cases, we first stop at the `main` function, as the initialization before that mostly consists of large memory clearing and memory copy loops that initialize the memory, for example, with zeros, and then copy the pre-existing global data to the memory location expected by the application. We identify the address of the `main` function though debug symbols in the firmware's *ELF* file. (1) At the starting spot (e.g., `main`), we halt the execution and create a snapshot. The HOEDUR execution engine provides the snapshotting mechanism and keeps track of the program's memory and execution state, which can be restored later. (2) Next, we iterate through all memory accesses performed by the firmware in the basic block currently halted at (*current BB*). From the execution trace, we know the exact PC, memory address, access size, and the value read. (3) We iterate through each accessed *bit*. If the bit is located in a text section, e.g., if it contains memory to be executed, we apply the bitflip immediately to the memory and clear QEMU's translation buffer, which keeps a cache of QEMU's Just-in-time (JIT) engine results. Since we change the instructions, we have to force QEMU to regenerate its JIT code. We rely on the assumption that from the start of the basic block until the execution of the instruction, the code does not change. If the bit is located in a data region, we do not apply the bitflip immediately; rather, we delay its injection until the actual execution of the targeted load instruction. In some scenarios, an address is written to before being read in a single basic block, for example, when pushing to the stack at a function's preamble, before returning and popping the register again, which triggers a memory read.

Having injected the bitflip into the *current BB*, we continue and track the program's execution, which we detail in the next section. After the execution, we restore the previously created snapshot and clear the injected bitflip. At this point, we have fully restored the program's state from before the bitflip. Then, our access iteration repeats the process for the next bit.

After concluding all bitflips for the *current BB*, we move to the next BB. During this step, the program's execution must diverge from our reference trace. Otherwise, we report an error in the experiment. Upon arriving at the next BB, we overwrite our previous snapshot and restart our bitflip loop. This way, we move through the entire program, through each access, and each bit in each access, and we test the behavior for each possible bitflip. Additionally, during these tests, we step through the program in an untainted manner (i.e., without bitflips present) once and check that it does not deviate from the reference, proving that the execution behaves as expected.

5.2.2. Execution Tracking. After injecting a bitflip, the target program is executed to track for differences in writes. Before the first execution, we extracted the expected order of BBs and the exact MMIO write and read values. This allows us to constantly check for any deviation during the program's execution. For example, if, during the execution, the target moves to a BB that diverges from the reference, we record a desynchronization (*BB desync*). We only record the first BB desync since most likely all subsequent BBs also differ, since the program execution path changed. However, we do not stop the program execution; instead, we keep it running to observe the changed behavior.

Further, we track each *MMIO read desync*, matching the MMIO address and PC address that attempted the read, and the access size and we record the first desynchronization. We also record a *desync* if the program attempts to read more MMIO values than in the trace. However, in this case, we terminate the execution since our execution trace cannot generate new input. Similarly, we track *MMIO write desyncs*, compare the value being written, and stop if more writes occur than expected.

In conclusion, RADSIM replays the input from fuzzing the *plain* target flavor on the *hardened* binaries to create the same execution paths. The *plain* and *hardened* flavors are fault-injected with every possible single bitflip, and the outcomes are recorded.

6. Evaluation

We evaluate the impact of SEEs on exploit mitigation techniques using our RADSIM prototype. We first evaluate the coverage of our test cases, forming the foundation for

TABLE 3: Statistics of the hardened binaries: Number of basic blocks, instructions, functions, and `.text` section size. In parenthesis: The ratio compared to *plain*.

Flavor	Basic Blocks	Instructions	`.text` size	Functions
ORESAT				
plain	8,023 (1.00)	43,276 (1.00)	126,930 (1.00)	1,269 (1.00)
canary-all	9,832 (1.23)	57,708 (1.33)	166,136 (1.31)	1,269 (1.00)
canary-strong	8,250 (1.03)	44,723 (1.03)	131,186 (1.03)	1,283 (1.01)
cfi-icall	8,249 (1.03)	43,681 (1.01)	128,760 (1.01)	1,344 (1.06)
san-func	8,743 (1.09)	46,743 (1.08)	142,410 (1.12)	1,316 (1.04)
ACUBESAT				
plain	9,405 (1.00)	52,713 (1.00)	157,060 (1.00)	2,925 (1.00)
canary-all	13,586 (1.44)	95,409 (1.81)	262,136 (1.67)	2,898 (0.99)
canary-strong	9,712 (1.03)	56,316 (1.07)	166,920 (1.06)	2,905 (0.99)
cfi	9,473 (1.01)	52,828 (1.00)	159,664 (1.02)	2,961 (1.01)
cfi-icall	9,728 (1.03)	53,085 (1.01)	159,236 (1.01)	2,963 (1.01)
san-func	9,545 (1.01)	54,220 (1.03)	163,648 (1.04)	2,953 (1.01)

our fault injection campaigns. Next, we discuss the flavored binaries and compare their general statistics. In our first analysis, we compare the raw outcomes of all our fault injection experiments and present the different ratios between the outcome types. We then calculate a total failure rate by resolving our time-domain assumption (cf. Section 4). Finally, we compare the localization of faults between non-hardened and hardened targets to investigate the primary source of faults.

6.1. Test Case Coverage

We start by evaluating the foundation of our previous steps: the initial fuzzing runs, which we use as test cases for the fault injection. We fuzzed ORESAT on 196 cores for 48 hours, amounting to 9,408 core hours. Further, we fuzzed ACUBESAT on 196 cores for 60 hours since we observed more coverage and the binary has more basic blocks overall, resulting in 11,760 core hours.

After fuzzing ORESAT, we achieved a coverage of 3,352 (42%) BBs out of 8,023 BBs; for ACUBESAT, we achieved a coverage of 5,869 (62%) of 9,405 BBs. The coverage is computed over the entire binary, including the operating systems, hardware abstraction layer, and other components not directly developed by the satellite team. Hence, the binary also contains portions of code that are unreachable in any fuzzing scenario. As usual in fuzzing scenarios without instrumentation, estimating the reachable code from the total code is impossible.

6.2. Targets and Flavors

Table 3 provides an overview of the general characteristics of the target binaries examined, namely ORESAT and ACUBESAT, which we picked as representative small satellite firmware implementations in Section 2.3. Both projects utilize the ARM Cortex-M architecture and operate on real-time operating systems, ACUBESAT on FreeRTOS and ORESAT on ChibiOS. To evaluate the resilience of these binaries, we compiled each firmware in several configurations, or *flavors*, each applying a distinct combination of exploit mitigations. The configurations examined are:

plain. This baseline configuration contains no exploit mitigation techniques, serving as a reference to assess the overhead and structural changes introduced by the mitigations.

canary-all. Includes stack canaries for all functions (`-fstack-protector-all`), regardless of whether they contain a stack buffer. This configuration represents the maximum protection achievable through stack canaries, as it applies uniformly to all functions. It is designed to evaluate the highest possible impact that canaries or similar mitigations may have on binary structure and runtime behavior.

canary-strong. Adds canaries to functions with an existing stack, even if there is no stack buffer but simply stack usage (`-fstack-protector-strong`). This selective application allows for a more optimized performance profile compared to the full protection of *canary-all*, as only functions with stack interactions are protected.

cfi-icall. Introduces CFI using a static check to verify whether a given function may legally be called from a particular call site (`-fsanitize=cfi-icall`). This flavor specifically addresses indirect function calls via function pointers, as ORESAT is written in C and does not use C++ vtable calls. ACUBESAT, on the other hand, which has parts in C++, could theoretically benefit from additional CFI checks for virtual calls, but this is not explored in the current configurations for comparability between the two targets.

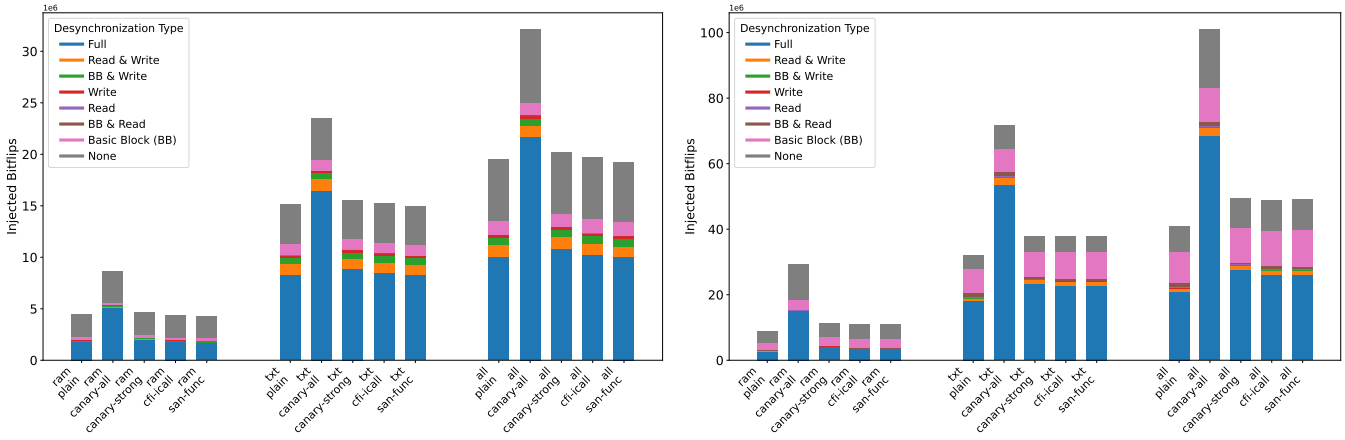
san-func. Introduces more comprehensive control flow protection compared to *cfi-icall* by dynamically checking function pointers' types using tags embedded within each function body (`-fsanitize=function`).

In addition, we compiled all binaries with `-flto`, which enables Link Time Optimization; as this is required for the CFI and function sanitization options, we included them in all flavors for increased comparability. In the same spirit, we also added `-fno-inline` to prevent the compiler from inlining functions, as this behavior might change based on the mitigations. We chose the optimization level `-O3` for all builds.

As expected, the *canary-all* targets increase the basic block counts significantly by 23% and 44%, respectively; for the other flavors, we generally observe a 1%–10% increase.

6.3. Stacked Outcomes

For our first experiment, we track the different outcomes of each fault-injected run, as shown in Figure 2. We executed RADSIM on 75 test cases for ORESAT and 35 for ACUBESAT, which we selected according to our coverage maximization strategy (cf. Section 5.1.3). We reduced the number of test cases compared to ORESAT due to the longer average execution time. The test cases result in a total of 8 billion bitflip experiments for ORESAT and 13 billion for ACUBESAT. During each execution, we recorded three different binary outcome states, where one of the following



(a) ORESAT. (b) ACUBESAT.

Figure 2: Stacked outcomes of an average fault injection campaign for both target satellites. We tracked three potential desynchronization types for each run, where the fault-injected run can differ from the reference, and we show all combinations.

criteria varies from the reference trace: Control flow path (here: basic blocks), MMIO reads, or MMIO writes. Three binary states result in eight possible combinations, which equates to the eight labels in the figure, where *full* means that all three are desynchronized and *none* implies that no change in behavior was observed. We accumulated the eight categories for each input and calculated the arithmetic mean over all inputs. Since we use total count here, the Y-axis refers to the average number of outcomes in each category, and the top of the bar marks the total number of bitflips performed in that category. For example, for *canary-all–all*, we performed 32 million fault injection runs on average per input in ORESAT; hence the Y-Axis notes the total count of outcomes in millions (1e6). We tested each flavor, and we separately report the numbers for *RAM*, where we only injected into load operations, *text*, where we only inject into instructions, and *all*, combining both.

As anticipated, RAM exhibits fewer bitflip-induced faults, given that not all instructions interact with RAM. Further, the most dominant colors are blue (full desync) and gray (no desync). In most cases, a desync correlates with a behavior change propagating to all aspects, i.e., different reads, writes, and control flows. Next, we can see that *canary-all* has the most significant overall bitflip counts, which correlates with the increase in `.text` section size from Table 3. Similarly, we can see that the other flavors roughly follow the `.text` section size increases in the total number of bitflips, except *san-func*. *san-func* places tags before every function for dynamic type checking; these tags take memory but are not used for almost any function. Hence, the change in `.text` section size does not materialize in a different bitflip count.

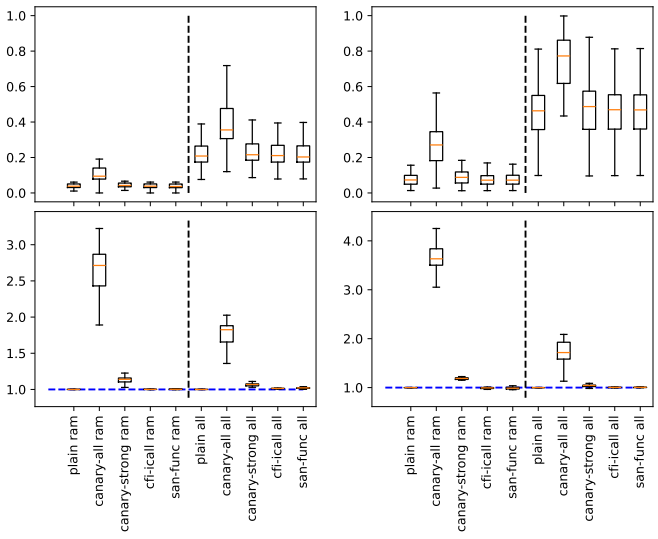
Looking at Figure 2, we cannot see any outlier where the number of bitflips or the ratios significantly deviate from the plain flavor, except the previously explained ones. This first hints towards the hardening techniques not impacting bitflip susceptibility beyond their code size increase. However, these numbers only describe all possible outcomes a single bitflip could cause in the program. Simply, they explain what happens if a bit in memory flips and if the program is reading it at the time of flipping, not earlier or later.

6.4. Overall Impact Study

In the subsequent evaluation, we want to estimate the overall impact of exploit mitigation features on the reliability of the satellite firmware. To do this, we need to recall that radiation can cause any bit in memory (spatial) to flip at any time (temporal); there are two dimensions to consider when calculating the impact of a single, random bitflip. Making this value space usable for analysis, RADSIM follows the program's path through this spatial-temporal domain and only tests flips that affect the program, effectively reducing the value space to one dimension (cf. Section 4). This results in a probability for an outcome (i.e., the desync type), given that the random bitflip hits the program's path.

Now, we separate the spatial and temporal domain to include the complete value domain and calculate the failure probability of full program runs. This metric also accounts for additional runtime induced by exploit mitigation features, showing the overall impact. In other words, if a bitflip occurs at a specific place (spatial), what is the probability that the program fails due to that bitflip.

$$
\begin{aligned}
&P_{fault}(program) = 1 - P_{succ}(program) \\
&= 1 - \prod_{t}^{n} (P_{succ}(BB_t)) = 1 - \prod_{t}^{n} (1 - P_{fault}(BB_t)) \\
&= 1 - \prod_{t}^{n} \left(1 - \frac{bits_{fault,t}}{bits_{total}}\right)
\end{aligned}
$$



(a) ORESAT (b) ACUBESAT

Figure 3: Failure probability when assuming a 5 MB chip size and one random bitflip at every point in time. The bottom chart shows the failure probability normalized to the plain binary.

To calculate this value, we must first define what the condition is for our program to *fail*. We consider any run a fault run if the MMIO writes something different from the reference run because if a program writes other values to peripherals, it most likely causes some observable malfunction. This is akin to observing the result of the calculation. Hence, any run outcome that includes an MMIO write desync is considered a fault run. Using our data, we can calculate the probability $P_{fault}(program)$ of a program failing, where $bits_{fault,t}$ describes the number of flipped bits that lead to a fault for a given basic block. In our implementation, we mention that we step through each basic block and test each possible bitflip. We consider each BB a unit in time and the number of bits that lead to a fault when flipped, the faulting bits at that point in time. For the total available bits $bits_{total}$, we have to assume the total memory size. If we assume a chip size of 5 MB to fit the small firmware and one random bitflip at every point in time (basic block), we get a failure distribution over different program inputs as shown in Figure 3.

The top plots show the failure probability for each flavor, given that a bitflip occurs somewhere in the memory. The bottom plots show the ratio of the flavors compared to plain. We again split the data into *RAM* only inject, where we only inject faults into load operations, and *all* flips, where we include both RAM and `.text` flips. From the ratio plots, we can see that *canary-all* induces a sizable impact on the bitflip resilience by making the program 2.7 resp. 3.6 times more likely to malfunction when bitflips in RAM occur. However, using the weaker *canary-strong* option leads to an increase of only 14% for ORESAT and 19% for ACUBESAT. Further, *cfi-icall* and *san-func* only come with marginally higher impacts compared to plain, with a 1.4% and 2.1% increase for ORESAT and 0.4% and 0.5% for ACUBESAT.

6.5. Fault Localization

The results of Section 6.4 show an increased fault probability for hardened binaries, particularly if stack canaries are enabled. To further analyze the cause of this difference, we compare the fault probabilities as depicted in Figure 4.

We select two hardened binary flavors for Figure 4. The top and bottom charts in this figure show the fault probability of each basic block grouped by function. Each vertical bar represents one function, and each dot indicates the probability of a fault due to a bitflip in this block. The data is accumulated over all runs. The top chart shows the unhardened, plain binary and the bottom one shows the hardened flavor. The chart in between shows the difference between the plain and hardened charts, adjusted for basic block alignment and with noise reduced. In detail, the alignment utilizes the fault probability of each function's basic blocks to identify added blocks in the hardened function body. It then aligns the values of the plain and hardened functions to minimize the differences while strictly keeping the basic block order. The result is that an additional basic block only shows as one different dot instead of shifting all the following blocks and making all remaining dots differ. Lastly, we hide small differences from the difference chart to make larger differences visible. This accounts for the noise introduced by the compiler optimizations for the different binary flavors.

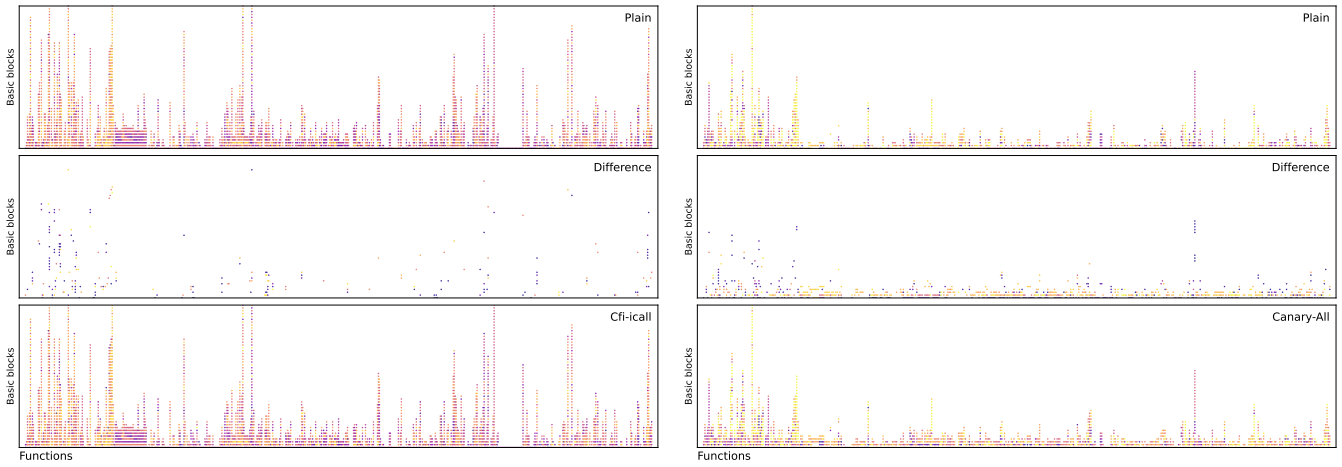
Figure 4a demonstrates the rather small impact of CFI on the fault probability. Differences are sparse and evenly distributed over the functions' basic blocks. Figure 4b shows the impact of *canary-all*. This figure excludes the bitflips to instructions, i.e., the `.text` section, to highlight the impact of the stack canaries stored on the stack. This chart clearly shows an increased fault probability throughout all functions. Further, the additional faults are primarily located at the bottom of the chart during the functions' prologue, where the canary is read from a global variable and stored on the stack. From the charts, we can conclude that we can locate the faults, and they appear as expected.

7. Discussion

Our results and approach warrant further discussion. We first examine the relationship between failure probability and code size increase. Next, we put our results into a real-world context and discuss the fault reason metric used in our evaluation. Finally, we address the challenges arising from evaluating academic hardening techniques, the implication of radiation hardening, as well as the limitations of our approach along with potential directions for future work.

7.1. Failure Probability vs. Code Size

Our results from Section 6.4 show that stack canaries impact the failure probability far greater than code size



(a) ORESAT: CFI-icall.

(b) ORESAT: Canary-All, excluding flips in `.text`.

Figure 4: Fault probability per basic block grouped by function. Each vertical bar depicts a function, of which each dot represents the fault probability of bitflips in that block. The first block per function is at the bottom; lighter colors indicate a higher fault probability. Comparing the unhardened and hardened versions reveals locations impacted by mitigations.

increases alone can explain. For example, between *plain* and *canary-all*, code sizes increased by 31% (ORESAT) and 67% (ACUBESAT). However, as shown in Figure 3, the corresponding failure probabilities increased by 171% and 263% (2.71 and 3.63), respectively. However, *canary-all* is unlikely to be deployed in real-world scenarios. Instead, *canary-strong* is a more realistic candidate: Here, the failure probability increases by 14% and 19%. In practice, this means that stack canaries alone increase the probability of a satellite malfunction due to a single event error by a value between 14% to 19%. Such malfunctions could lead to crashes (reducing system uptime) or more subtle errors, such as ill-formed instructions for peripheral accesses (MMIO writes), which in turn could lead to permanent damage. Whether this 14% to 19% increase in failure probability is critical for satellite developers depends on the specific project and requires careful risk estimation as well as engineering judgment. If developers rule out stack canaries due to this risk, there is a clear need for radiation-resistant stack canary alternatives for space systems. However, suppose this increase in failure probability is deemed acceptable. In that case, as shown by our experiments, there is no fundamental *technical* reason preventing wider adoption of stack canaries in space systems.

7.2. Real-World Context

Deriving specific real-world crash rates from our results is difficult and prone to misconceptions. For this reason, our evaluation only attempts to show the difference to a baseline. For example, in Section 6.4, we show that we can observe a 14% increase in crashes related to stack canaries when observing the total memory, i.e., both RAM and text. Hence, we leave the real-world crash rate to future research. Ultimately, precisely modeling real-world SEUs involves elaborate radiation modeling, i.e., the Sun's activity, precise hardware models (for example, using FreePDK [8], [50]), and more, which is beyond the scope of this paper. In addition, previous research states that instruction memory is technically read-only, allowing for elaborate error detection and correction [11], [48], [64]. However, our survey of satellite systems did not give us reason to assume a widespread deployment of such measures.

Thus, in our research, we follow the approach of dependable systems research that models bitflips in software by stating failure probabilities and rates that assume a bitflip occurs while leaving the actual probability for a bitflip to a different research field. Other research investigates actual bitflip probabilities depending on radiation and the incoming particles, for example, by placing processor and memory in particle accelerators [12], [24], [27], [57], [68].

7.3. Fault Reasons

During our overall impact study (cf. Section 6.4), we studied the impact of single event errors on the program outcomes. We divided the program outcomes into *success* and *failure* cases, and we categorized outcomes based on MMIO write desynchronization. An MMIO write desync occurs if the fault-injected program's MMIO write does not match the reference trace, which has been created without fault injection. We chose this metric, as MMIO writes are used to communicate with external peripherals and devices, and changes in MMIO writes are the strongest indicator if the satellite's OBC gives faulty instructions to other devices, potentially leading to malfunction or lasting damage. After a peripheral is given the wrong instructions, it is likely difficult for operators to figure out the problem based solely on the observation of its unexpected behavior. An alternative metric to MMIO write desyncs could have been tracking *software-*

only desyncs, e.g., different execution paths (that is, desyncs of executed basic blocks). However, there is a (small) chance these resolve themselves if the thread that is desynchronized terminates or receives new inputs. We conservatively opted to only track errors that result in consequences for the hardware. However, this distinction between hardware and software desynchronizations is ultimately not critical, as can be seen in Figure 2: In most cases, either all or none of the criteria desynchronize, and the relative share of the outcomes seems to be the same across a satellite. Therefore, changing the categorization of fault reasons from MMIO writes to a software-based metric only changes the absolute numbers. For example, if we include *basic block desyncs*, the total number of faults rises slightly—but the share across categories remains equal.

7.4. Academic Hardening Techniques

In recent years, academic research has presented a series of exploit mitigation techniques beyond the hardening methods explored in this work [4], [42], [45], [63]. Initially, we considered testing a series of academic hardening methods as well, which coincided well with recent work from 2024 by Tan et al. [69] that provides a systematization of knowledge on the (academic) progress of exploit mitigations for ARM Cortex-A & Cortex-M. When investigating the individual research papers discussing novel hardening techniques, we noticed that the published prototypes are scattered over many years of research. Consequently, their prototype implementations—most in the form of LLVM passes—target widely different versions of LLVM, partially going as far back as LLVM 3.9 [21]. While this poses no problem for research on exploit mitigations, it presents a problem for our fault injection-based approach. Different compiler versions require different *plain* flavor baselines for a meaningful evaluation, making comparisons increasingly complex. We considered porting proposed mitigations to the current LLVM version but deemed this unfeasible.

Furthermore, many techniques do not inherently conflict with radiation. For example, address space layout randomization (ASLR) and executable-only memory (XOM) do not contain sensible values like stack canaries or function sanitization tags that are susceptible to radiation. For example, ASLR is managed at an OS level; while not straightforward for MMU-free RTOS systems, there are proposed solutions [45], [63], but the actual binary is loaded at different addresses without changes to the binary. Similarly, while XOM is not natively supported by the ARM Cortex-M memory protection unit, research has also come up with solutions [42], [62].

7.5. Implications for Rad-Hardened Designs

Our RADSIM engine calculates the probability that a target program fails given that a fault has occurred. In other words, we compute the conditional probability of a fault causing a failure. As far as we can tell, radiation-hardened processors primarily reduce the likelihood of a particle strike causing a fault in the first place. Therefore, the failure mitigation provided by radiation-hardened hardware operates at a different level than the one our work studies, and our results also apply to radiation-hardened designs. These techniques merely ensure that the likelihood of encountering a fault is significantly reduced; if a fault occur, our observations hold likewise. The key difference lies in the overall probability of failure due to a particle strike, which is lower in radiation-hardened systems but follows the same underlying mechanisms.

7.6. Limitations

While our prototype only works for ARM Cortex-M, this is purely an engineering limitation stemming from the underlying HOEDUR firmware fuzzer we used as foundation [61]. Coincidentally, fuzzers have many of the same requirements as our fault injection approach, such as precisely controlling and stopping execution, controlling input and output, and achieving high execution throughput. There is no conceptual limit to applying our approach to ARM Cortex-A or AVR32, which would be according to our hardware survey (cf. Section 2.1). Hence, while not conceptually limited, the engineering effort to implement new architectures, such as radiation-hardened architectures, is considerable. Previous research on satellite security has yielded a QEMU implementation for AVR32 [75], [76].

Another limitation of our prototype is that it does not inject faults into registers. From a technical perspective, it is trivial to also iterate through each register access and test each single bitflip, as we did with instructions and memory accesses. However, since each instruction works with at least one register, i.e., 32 bits, this quickly increases the computing effort by a huge margin. Further, register values are extremely short-lived, often only for a few instructions. In practice, this usually means that inducing a bitflip during a memory load has the same outcome as a bitflip to the respective register value in the subsequent instructions. Nevertheless, there are approaches [10] to analyze the assembly and reduce the fault injection to the bits that are actually utilized, but they still rely on random sampling during the evaluation.

8. Conclusion

This work explored the impact of radiation-induced bitflips, specifically Single Event Errors (SEEs), on the resilience of exploit mitigations in satellite firmware. In the first step, we performed an extensive market survey of 381 satellite designs and concluded that only 8.5% deploy radiation-hardened processors. Subsequently, we investigated the impact of SEEs on the exploit hardening technique by proposing a novel binary-agnostic and deterministic approach for injecting bitflips, which exhaustively tests every single bitflip option for both `.text` and memory reads. We develop RADSIM and systematically evaluate the radiation impact on stack canaries, control flow integrity, and function sanitization using 21 billion bitflip experiments.

Our findings reveal that all tested hardening techniques increase the probability of a program malfunctioning. Additionally, we showed that the increase in fault probability for stack canaries exceeds the increase explained through code size increase, showing that the canaries themselves measurably increase a system fault probability in radiation-heavy environments. However, for other defenses, such as CFI, the primary factor for increased failure probability is code size.

Acknowledgement This work was funded by the European Research Council (ERC) under the consolidator grant RS^3 (101045669), the German Federal Ministry of Education and Research (BMBF) under the grant CPSec (16KIS1899), and Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972. The work was partially supported by the MKW-NRW research training group SecHuman.

References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[2] Ali Abbasi, Jos Wetzels, Thorsten Holz, and Sandro Etalle. Challenges in designing exploit mitigations for deeply embedded systems. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[3] AcubeSAT Team. AcubeSAT OBC Software, 2024. https://gitlab.com/acubesat/obc/obc-software.

[4] Naif Saleh Almakhdhub, Abraham A Clements, Saurabh Bagchi, and Mathias Payer. μRAI: Securing embedded systems with return address integrity. In *Network and Distributed Systems Security (NDSS) Symposium*, 2020.

[5] Rui Almeida, Vitor Silva, and Jorge Cabral. Virtualized fault injection framework for iso 26262-compliant digital component hardware faults. *Electronics*, 2024.

[6] Jianfeng An, Hongjun You, Fengqing Xie, Yuanlin Yang, and Jinhua Sun. Fig-qemu: A fault inject platform supporting full system simulation. In *International Conference on Dependable Systems and Their Applications (DSA)*, 2020.

[7] Atmel Corporatio. AVR32006 : Getting started with GCC for AVR32. https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ApplicationNotes/ApplicationNotes/doc32074.pdf, 2007.

[8] Sarah Azimi, Corrado De Sio, Daniele Rizzieri, and Luca Sterpone. Analysis of single event effects on embedded processor. *Electronics*, 2021.

[9] Christoph Bader. On requirements & concepts for tt&c link key management. In *2nd Workshop on the Security of Space and Satellite Systems (SpaceSec)*, 2024.

[10] Raul Barbosa, Jonny Vinter, Peter Folkesson, and Johan Karlsson. Assembly-level pre-injection analysis for improving fault injection efficiency. In *European Dependable Computing Conference*, 2005.

[11] Michael Barr. *Programming embedded systems in C and C++*. O'Reilly Media, 1999.

[12] Fabio Benevenuti, Leonardo Gobatto, R Possamai Bastos, José Rodrigo Azambuja, and Fernanda Lima Kastensmidt. Assessing the impacts of radiation-induced soft errors on arm cortex-m microprocessors. *IEEE Transactions on Nuclear Science*, 2024.

[13] Youcef Bentoutou, El-Habib Bensikaddour, Nasreddine Taleb, and Nacer Bounoua. An improved image encryption algorithm for satellite applications. *Advances in Space Research*, 2020.

[14] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 2011.

[15] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Compiler-implemented differential checksums: Effective detection and correction of transient and permanent memory errors. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2023.

[16] Panagiotis Bountzioukas, Georgios Kikas, Christoforos Tsiolakis, Dimitrios Stoupis, Eleftheria Chatziargyriou, Alkis Hatzopoulos, Vasiliki Kourampa-Gottfroh, Ilektra Karakosta-Amarantidou, Aggelos Mavropoulos, Ioannis-Nikolaos Komis, et al. The evolution from design to verification of the antenna system and mechanisms in the acubesat mission. In *International Astronautical Congress (IAC)*, 2023.

[17] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys*, 50(1):1–33, 2017.

[18] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining Light on Shadow Stacks. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[19] Athanasios Chatzidimitriou, Pablo Bodmann, George Papadimitriou, Dimitris Gizopoulos, and Paolo Rech. Demystifying soft error assessment strategies on arm cpus: Microarchitectural fault injection vs. neutron beam experiments. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019.

[20] Sławomir Chyłek and Marcin Goliszewski. Qemu-based fault injection framework. *Studia Informatica*, 33(4):25–42, 2012.

[21] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

[22] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security Symposium*, 1998.

[23] Paul E Dodd and Lloyd W Massengill. Basic mechanisms and modeling of single-event upset in digital microelectronics. *IEEE Transactions on Nuclear Science*, 2003.

[24] Paul E Dodd, James R Schwank, Marty R Shaneyfelt, James Andrew Felix, P Paillet, V Ferlet-Cavrois, J Baggio, RA Reed, KM Warren, RA Weller, et al. Impact of heavy ion energy and nuclear interactions on single-event upset and latchup in integrated circuits. *IEEE Transactions on Nuclear Science*, pages 2303–2311, 2007.

[25] Boyang Du, Luca Sterpone, Sarah Azimi, David Merodio Codinachs, Véronique Ferlet-Cavrois, Cesar Boatella Polo, Rubén García Alía, Maria Kastriotou, and Páblo Fernandez-Martínez. Ultrahigh energy heavy ion test beam on xilinx kintex-7 sram-based fpga. *IEEE Transactions on Nuclear Science*, 2019.

[26] Davide Ferraretto and Graziano Pravadelli. Simulation-based fault injection with qemu for speeding-up dependability analysis of embedded software. *Journal of Electronic Testing*, 2016.

[27] Jonas Gava, Nicolas Moura, Joaquim Lucena, Vinìcius da Rocha, Rafael Garibotti, Ney Calazans, Sergio Cuenca-Asensi, Rodrigo Possamai Bastos, Ricardo Reis, and Luciano Ost. Assessment of radiation-induced soft errors on lightweight cryptography algorithms running on a resource-constrained device. *IEEE Transactions on Nuclear Science*, 2023.

[28] GCC Team. x86 Options. https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html, 2024.

[29] Giorgis Georgakoudis, Ignacio Laguna, Dimitrios S Nikolopoulos, and Martin Schulz. Refine: Realistic fault injection via compiler-based instrumentation for accuracy, portability and speed. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[30] Andrew Greenberg, David Lay, Glenn Lebrasseur, Emma Levy, Ryan Medick, Hayden Reinhold, Risto Rushford, and Miles Simpson. Oresat: A student team-based approach to an inexpensive, open, and modular (1-3u) cubesat bus. In *2021 Small Satellite Conference*, 2021.

[31] Tamara Gutierrez, Alexandre Bergel, Carlos E Gonzalez, Camilo J Rojas, and Marcos A Diaz. Systematic fuzz testing techniques on a nanosatellite flight software for agile mission development. *IEEE Access*, 2021.

[32] Alex DP Hands, Keith A Ryden, Nigel P Meredith, Sarah A Glauert, and Richard B Horne. Radiation effects on satellites during extreme space weather events. *Space Weather*, 2018.

[33] Florian Hauschild, Kathrin Garb, Lukas Auer, Bodo Selmke, and Johannes Obermaier. Archie: A qemu-based framework for architecture-independent evaluation of faults. In *Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, 2021.

[34] Andrea Höller, Gerhard Schönfelder, Nermin Kajtazovic, Tobias Rauter, and Christian Kreiner. Fies: a fault injection framework for the evaluation of self-tests for cots-based safety-critical systems. In *International Microprocessor Test and Verification Workshop*, 2014.

[35] M. Hsueh, T. Tsai, and R. Iyer. Fault injection techniques and tools. *Computer*, 1997.

[36] Samuel Jackson, Scott Kerlin, and Jeremy Straub. Poster: Implementing and testing a novel chaotic cryptosystem for use in small satellites. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[37] Oltjon Kodheli, Eva Lagunas, Nicola Maturo, Shree Krishna Sharma, Bhavani Shankar, Jesus Fabian Mendoza Montoya, Juan Carlos Merlano Duncan, Danilo Spano, Symeon Chatzinotas, Steven Kisseleff, et al. Satellite Communications in the New Space Era: A Survey and Future Challenges. *IEEE Communications Surveys & Tutorials*, 2020.

[38] Erik Kulu. Nanosats Database. https://www.nanosats.eu/, 2019.

[39] Erik Kulu. Satellite Constellations-2021 Industry Survey and Trends. In *Annual Small Satellite Conference*, 2021.

[40] Erik Kulu. Nanosatellite launch forecasts-track record and latest prediction. In *Small Satellite Conference*, 2022.

[41] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[42] Donghyun Kwon, Jangseop Shin, Giyeol Kim, Byoungyoung Lee, Yeongpil Cho, and Yunheung Paek. uXOM: Efficient execute-only memory on arm. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[43] Xiaodong Li, Sarita V Adve, Pradip Bose, and Jude A Rivers. Architecture-level soft error analysis: Examining the limits of common assumptions. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2007.

[44] Qining Lu, Mostafa Farahani, Jiesheng Wei, Anna Thomas, and Karthik Pattabiraman. Llfi: An intermediate code-level fault injection tool for hardware faults. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, 2015.

[45] Lan Luo, Xinhui Shao, Zhen Ling, Huaiyu Yan, Yumeng Wei, and Xinwen Fu. faslr: Function-based aslr via trustzone-m and mpu for resource-constrained iot systems. *IEEE Internet of Things Journal*, 2022.

[46] Katrina Manson. The Satellite Hack Everyone is Finally Talking About. *Bloomberg*, 2023. https://www.bloomberg.com/features/2023-russia-viasat-hack-ukraine/#xj4y7vzkg.

[47] R. Natella, Domenico Cotroneo, and H. Madeira. Assessing dependability with software fault injection. *ACM Computing Surveys*, 2016.

[48] Bogdan Nicolescu, Raoul Velazco, and Matteo Sonza Reorda. Effectiveness and limitations of various software techniques for" soft error" detection: a comparative study. In *International On-Line Testing Workshop*. IEEE, 2001.

[49] Christoph Noeldeke, Maximilian Boettcher, Ulrich Mohr, Steffen Gaisser, Mikel Alvarez Rua, Jens Eickhoff, Mike Leslie, Matt Von Thun, Sabine Klinkner, and Renuganth Varatharajoo. Single event upset investigations on the "flying laptop" satellite mission. *Advances in Space Research*, 2021.

[50] Semiu A Olowogemo, Hao Qiu, Bor-Tyng Lin, William H Robinson, and Daniel B Limbrick. Model-based analysis of single-event upset (seu) vulnerability of 6t sram using finfet technologies. In *2022 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2022.

[51] OreSat Team. OreSat C3 Software, 2024. https://github.com/oresat/oresat-c3-software.

[52] George Papadimitriou and Dimitris Gizopoulos. Demystifying the system vulnerability stack: Transient fault effects across the layers. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[53] Konstantinos Parasyris, Georgios Tziantzoulis, Christos D Antonopoulos, and Nikolaos Bellas. Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[54] James Pavur, Daniel Moser, Vincent Lenders, and Ivan Martinovic. Secrets in the Sky: On Privacy and Infrastructure Security in DVB-S Satellite Broadband. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2019.

[55] James Pavur, Daniel Moser, Martin Strohmeier, Vincent Lenders, and Ivan Martinovic. A Tale of Sea and Sky: On the Security of Maritime VSAT Communications. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[56] Clémence Poirier. Hacking the cosmos: Cyber operations against the space sector: A case study from the war in ukraine. *CSS Cyberdefense Reports*, 2024.

[57] RA Reed, MA Carts, PW Marshall, CJ Marshall, O Musseau, PJ McNulty, DR Roth, S Buchner, J Melinger, and T Corbiere. Heavy ion and proton-induced single event multiple upset. *IEEE Transactions on Nuclear Science*, 1997.

[58] Behrooz Sangchoolie, Karthik Pattabiraman, and Johan Karlsson. An empirical study of the impact of single and multiple bit-flip errors in programs. *IEEE Transactions on Dependable and Secure Computing*, 2020.

[59] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using precise MMIO modeling for effective firmware fuzzing. In *USENIX Security Symposium*, 2022.

[60] Tobias Scharnowski, Felix Buchmann, Simon Wörner, and Thorsten Holz. A case study on fuzzing satellite firmware. In *1st Workshop on the Security of Space and Satellite Systems (SpaceSec)*, 2023.

[61] Tobias Scharnowski, Simon Wörner, Felix Buchmann, Nils Bars, Moritz Schloegel, and Thorsten Holz. Hoedur: Embedded Firmware Fuzzing using Multi-stream Inputs. In *USENIX Security Symposium*, 2023.

[62] Zhuojia Shen, Komail Dharsee, and John Criswell. Fast execute-only memory for embedded systems. In *2020 IEEE Secure Development (SecDev)*, 2020.

[63] Jiameng Shi, Le Guan, Wenqiang Li, Dayou Zhang, Ping Chen, and Ning Zhang. Harm: Hardware-assisted continuous re-randomization for microcontrollers. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2022.

[64] Philip P Shirvani, Nirmal R Saxena, and Edward J McCluskey. Software-implemented edac protection against seus. *IEEE Transactions on reliability*, 2000.

[65] Jacopo Sini, Massimo Violante, and Fabrizio Tronci. A novel iso 26262-compliant test bench to assess the diagnostic coverage of software hardening techniques against digital components random hardware failures. *Electronics*, 2022.

[66] Catherine Spivey and Emilio Gizzi. A modular, open source cubesat structure. In *AIAA Scitech 2021 Forum*, 2021.

[67] Vilas Sridharan, Jon Stearley, Nathan DeBardeleben, Sean Blanchard, and Sudhanva Gurumurthi. Feng shui of supercomputer memory: Positional effects in dram and sram faults. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[68] Lucas Antunes Tambara, Fernanda Lima Kastensmidt, Nilberto H Medina, Nemitala Added, Vitor AP Aguiar, Fernando Aguirre, Eduardo LA Macchione, and Marcilei AG Silveira. Heavy ions induced single event upsets testing of the 28 nm xilinx zynq-7000 all programmable soc. In *2015 IEEE Radiation Effects Data Workshop (REDW)*, 2015.

[69] Xi Tan, Zheyuan Ma, Sandro Pinto, Le Guan, Ning Zhang, Jun Xu, Zhiqiang Lin, Hongxin Hu, and Ziming Zhao. Sok: Where's the "up"?! a comprehensive (bottom-up) study on the security of arm cortex-m systems. In *USENIX WOOT Conference on Offensive Technologies (WOOT)*, 2024.

[70] The GCC Team. GCC Program Instrumentation Options. https://gcc.gnu.org/onlinedocs/gcc-6.3.0/gcc/Instrumentation-Options.html, 2024.

[71] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security Symposium*, 2014.

[72] Viasat Corporate. KA-SAT Network Cyber Attack Overview, 2022. https://www.viasat.com/about/newsroom/blog/ka-sat-network-cyber-attack-overview/.

[73] Jiesheng Wei, Anna Thomas, Guanpeng Li, and Karthik Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

[74] Johannes Willbold, Moritz Schloegel, Robin Bisping, Martin Strohmeier, Thorsten Holz, and Vincent Lenders. VSAsTer: Uncovering Inherent Security Issues in Current VSAT System Practices. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2024.

[75] Johannes Willbold, Moritz Schloegel, Florian Göhler, Tobias Scharnowski, Nils Bars, Simon Wörner, Nico Schiller, and Thorsten Holz. Scaling software security analysis to satellites: Automated fuzz testing and its unique challenges. In *IEEE Aerospace Conference*, 2024.

[76] Johannes Willbold, Moritz Schloegel, Manuel Vögele, Maximilian Gerhardt, Thorsten Holz, and Ali Abbasi. Space Odyssey: An Experimental Software Security Analysis of Satellites. In *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[77] Chunhua Yang, Chao Yang, Tao Peng, Xiaoyue Yang, and W. Gui. A fault-injection strategy for traction drive control systems. *IEEE Transactions on Industrial Electronics*, 2017.

Appendix A. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

This paper studies the impact of radiation-induced bitflips on the exploit mitigation techniques used by small satellite firmware. It first analyzes 381 small satellite designs to identify the prevalence of COTS hardware platforms. Next, it presents RADSIM, a new bitflip injection system, and demonstrates its use to evaluate two satellite firmware, each compiled with four exploit mitigation strategies.

A.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Establishes a New Research Direction
- Provides a Valuable Step Forward in an Established Field
- Other

A.3. Reasons for Acceptance

1) This paper identifies a key gap in security research on small satellite systems, which didn't explore the impact of space-specific factors on the deployment of traditional mitigation strategies.
2) The paper presents a systematic survey of small satellites that identifies a critical new finding: only 8.5% of the systems use radiation-hardened processors.
3) This paper examines cosmic radiation's impact on small satellite firmware that uses traditional exploit mitigation techniques with a newly developed tool for fault injection; the tool will be open-sourced to enable further research.

A.4. Noteworthy Concerns

1) Limitations of the paper's study/evaluation in: (1) the number (381) of satellite systems chosen, (2) the number of tested satellite firmware (two) and hardening techniques (four) [focused on software-based but not hardware-based], and (3) the number of test cases selected (75 test cases for ORESAT and 35 for ACUBESAT) to proceed with a comparison between plain and hardened code.