Mars Attacks!
Software Protection Against Space Radiation

Haoda Wang*
Columbia University
haoda.wang@columbia.edu

Steven Myint
Jet Propulsion Laboratory
California Institute of Technology
steven.myint@jpl.nasa.gov

Vandi Verma
Jet Propulsion Laboratory
California Institute of Technology
vandi.verma@jpl.nasa.gov

Yonatan Winetraub
CryptoSat
yonatan@cryptosat.io

Junfeng Yang
Columbia University
junfeng@cs.columbia.edu

Asaf Cidon
Columbia University
asaf.cidon@columbia.edu

ABSTRACT

Due to their low cost and the need to run computationally-intensive algorithms locally, satellites and spacecraft are increasingly employing off-the-shelf computing hardware. However, hardware in space is exposed to significantly higher amounts of radiation than on Earth, potentially destroying the hardware or causing it to output incorrect results. We envision that solely using software fault tolerance techniques, commodity hardware operating in space can achieve fault tolerance *equivalent* or close to expensive and slow radiation-hardened hardware. To achieve this goal, we need to address the two main radiation fault scenarios: hardware overheating and silent data corruption. We provide preliminary data on the effects of these errors, and introduce a set of techniques to address them. Enabling the full use of commodity hardware in space holds the promise of improving the compute capabilities and cost effectiveness of low-earth orbit satellites by orders of magnitude.

CCS CONCEPTS

• **Networks** → **Error detection and error correction**; • **Computer systems organization** → **Reliability**; • **Software and its engineering** → *Compilers*; *Operating systems*;

KEYWORDS

satellite computing, fault tolerance, radiation hardening

Specification	EnduroSat OBC	Snapdragon 801
Radiation-hardened	Yes	No
ISA	ARMv7E-M	ARMv7-A
Clock Speed	216MHz	2.5GHz
RAM	64MB ECC	2GB non-ECC
Storage	256MB Flash	32GB Flash
Cost	$10,000	$750

Table 1: Comparison between commodity and radiation hardened computers common to spacecraft.

ACM Reference Format:
Haoda Wang, Steven Myint, Vandi Verma, Yonatan Winetraub, Junfeng Yang, and Asaf Cidon. 2023. Mars Attacks! Software Protection Against Space Radiation. In *Proceedings of The 22nd ACM Workshop on Hot Topics in Networks (HotNets '23).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3626111.3628199

1 INTRODUCTION

The cost of launching payloads to space has sharply decreased, from $88K[1] per kilogram in 1981 on the Space Shuttle to just $1.4K on SpaceX's Falcon Heavy today [1]. Thus, private and government organizations are launching spacecraft at an exponentially increasing rate [2], aiming to create large satellite constellations that support a variety of use cases from Internet connectivity [3–5], real-time imaging [6, 7], blockchain processing [8, 9], to wireless power delivery [10]. Taking advantage of the decreased launch cost, constellations in low-earth orbit (LEO) such as Starlink have proven capable of providing significant performance and latency gains over traditional satellites in geostationary orbit [11]. As a consequence, LEO communications constellations have also attracted significant recent interest in the networking research community [11–17].

Minimizing the cost of each satellite in the constellation is a key concern to operators, and low-cost SmallSats below 1,200kg strike an ideal balance between cost and functionality [18]. However, the unique challenges of computing in

*Also with Jet Propulsion Laboratory, California Institute of Technology.

HotNets '23, November 28–29, 2023, Cambridge, MA, USA

ACM ISBN 979-8-4007-0415-4/23/11…$15.00
https://doi.org/10.1145/3626111.3628199

[1]Normalized to 2023 dollars.

a space environment restrict the choice of chips onboard satellites. Without the protection of Earth's atmosphere and magnetic field, missions traditionally employ specialized, radiation-hardened hardware. Due to the niche market for such chips, these computers are orders of magnitude more costly than their commodity counterparts [19]. This high cost is untenable for many emerging LEO use cases.

Making matters worse, as shown in Table 1, such chips boast woefully underpowered compute power, since radiation hardening requirements drive down clock speed and increase die areas [20]. Furthermore, recent advances in satellite communications such as 100Gbps data links [21] require compute capabilities far beyond those supported by hardened chips. Clearly, the demand for highly capable local compute in space cannot be met solely by radiation-hardened hardware [22]. Even worse, radiation-hardened hardware often requires non-mainstream operating systems and toolchains sometimes a decade out of date [23]. Missions are further restricted by thermal requirements to using low-power computers that do not generate much heat. These engineering decisions decrease developer productivity and greatly increase mission costs [24].

Thus, many SmallSat operators have decided to use Linux on commodity hardware (e.g. Raspberry Pi) due to cost and compute constraints [25–28]. Even far costlier space missions, such as flagship NASA missions, have started using commodity hardware due to its exponentially better performance. For example, the Ingenuity Mars Helicopter uses 2 Snapdragon 801s running Linux [29], and there are plans to use it for complex image processing tasks [30]. This trend of shifting from special-purpose to commodity hardware resembles how large-scale cloud infrastructure began on commodity servers instead of specialized HPC nodes [31, 32].

However, commodity hardware can encounter silent data corruption or even be damaged due to radiation errors [33]. This may cause outages or degradation of essential services [34] such as Internet or GPS [35]. Restoring these services can be very expensive, as new satellites need to be rebuilt and relaunched to replace the damaged ones.

Vision. Our vision is to democratize the use of commodity hardware in space by introducing *purely software-based fault tolerance* techniques to commodity computers, with the goal of matching the fault tolerance of their radiation-hardened counterparts. We hope to do so with minimal modifications to the host OS and without using custom hardware, minimizing developer friction [30]. This will further drive down the cost of launches and allow spacecraft to utilize state-of-the-art processors and accelerators (e.g. for onboard packet processing or network functions).

Error types. Two primary types of errors prevent the usage of commodity hardware in space networking infrastructure. The first error type is single-event latchups (SELs), which are localized short-circuits that can burn out the device, potentially destroying the computer [36]. These errors can occur at any time and may show up only as a very small increase in current draw, making them very difficult to detect. The second scenario is single-event upsets (SEUs), which are radiation-induced bit flips that can cause crashes, hangs, and silent data corruption [36]. Such errors can happen at any time and affect even protected memory, making them very hard to defend against at a software level. The common approach of redoing computations is often expensive due to power and thermal constraints in space. From conversations with satellite and spacecraft operators, these radiation errors are the primary barrier preventing spacecraft from replacing the aging radiation-hardened processors onboard with faster, cheaper commodity hardware [26, 30].

Previous efforts. Previous work has examined the risks of radiation errors on satellite Internet infrastructure [35, 37], but these have often centered on letting the affected satellite fail and having the network adapt to the failure, rather than catching the issue and preventing the failure in the first place. State-of-the-art software methods include setting a maximum current draw before power cycling the device and simply re-running target programs and ensuring the outputs match. However, these approaches are costly and treat the hardware and software as a black box.

Preliminary fault tolerance techniques. We note that not all missions have the same risk sensitivities and may be willing to trade some risk for higher performance. Thus, we design a set of techniques that allow flexible trade-offs between overhead and correctness guarantees.

We first introduce an SEL detection system that uses system metrics available to the OS to detect when current draw is higher than expected. Our main insight is that visibility into resource usage (e.g. CPU, memory, I/O) extracted via performance counters can significantly increase the detection rate of these errors with much less system overhead. We thus introduce a noninvasive method that uses these metrics to model expected current draw (§3.1).

Next, we introduce *tunable double modular redundancy*, a novel method to ensure correctness in programs vulnerable to SEUs. Our key idea is that not all SEUs are equal, and some bits will affect program outputs more than others. Therefore, protecting just the critical bits will provide strong correctness guarantees. We introduce a redundancy scheme using this idea that can be tuned to various accuracy and detection rates to adapt to different missions (§4.1). We also use this insight to create a metric for evaluating how SEUs may alter the final result of a computation (§4.2).

Finally, we note that missions need to ensure memory integrity since commodity computers often do not have hardware ECC. Our key idea is that spacecraft often do not make

use of dedicated onboard accelerators, as their use cases may not be applicable to spaceflight. Thus, we design a memory scrubber that can run on these underutilized accelerators, such as DSP coprocessors (§4.1).

We plan to validate these techniques in real space missions operated by our two partners: NASA-JPL and CryptoSat [26, 30]. In conclusion, we believe that by addressing the main causes of radiation errors with software fault tolerance techniques, commodity hardware can be safely deployed in space, thereby significantly reducing the cost and increasing the capabilities of satellites and spacecraft.

2 RADIATION HAZARDS IN SPACE

Ionizing radiation in space has three sources: galactic cosmic rays, solar radiation, and protons trapped in Earth's magnetic field [38]. Cosmic rays are fast, energetic particles originating from outside the Solar System. The Sun steadily releases charged particles in the solar wind and periodically releases stronger radiation bursts in solar particle events. Particles can also be trapped in Earth's magnetic field, forming Van Allen radiation belts around the planet.

Radiation on Earth. As Earth's atmosphere and magnetic field deflect or absorb most highly-charged radiation particles, radiation hardening is not a significant concern for computers on Earth, though cosmic rays and solar particle events occasionally cause errors [39]. However, satellites in orbit have less protection from Earth's atmosphere and are affected by trapped protons in the magnetosphere and solar particle events. Beyond Earth's orbit, computer systems require protection against cosmic rays and Solar radiation.

Radiation in Space. Cosmic rays, solar radiation, and trapped particles can all cause errors in chips, as modern commodity chips are not designed with radiation hardening in mind [40]. Shielding around the chip, the substances used in the die, and the spectrum of ionizing radiation encountered in the environment are all important factors when calculating the chances of a radiation error occurring [41]. Radiation can cause total ionizing dose effects, where the semiconductor structure of the computer degrades as it is exposed to radiation over time [42]. In contrast, our focus is on transient effects, which cause temporary errors.

While it is difficult to predict when these radiation errors will occur, such errors are the most common root cause of software errors on spacecraft [43]. Thus, it is imperative for space missions to understand how they manifest and defend against them. We next examine two primary types of radiation errors that spacecraft frequently encounter.

3 SINGLE EVENT LATCH-UPS

High-energy charged particles that impact transistor structures may induce a *single-event latch-up* (SEL), which is an erroneous transistor structure in the device that effectively acts as a short-circuit [44]. SELs dissipate a large concentration of energy on a few gates [45], generating excess heat that cannot be dissipated in the vacuum of space. This increases the temperature around these gates significantly and damages the chip by destroying the gate within around 3 minutes [26]. Fortunately, such an error can be fixed immediately by simply power cycling the affected device. However, detecting a latch-up in time before it permanently damages the device is not trivial, and consequently, latch-up errors have caused the loss of many commercial SmallSats [26, 46].

SEL-induced additional current draw may be less than 5 mA [47], which is negligible compared to current variations in modern CPUs due to power scaling, which, for example, can reach over 4.5 A on a Raspberry Pi. Therefore, naive detection solutions, such as triggering a reboot when a current threshold is exceeded, can incur many false positives due to natural variations in the processor's current draw. On the other hand, if the current threshold is set too high, the increase in false negatives risks the safety of the satellite.

3.1 Methods

Prior methods. Previous work in detecting and mitigating SELs use cheap current monitoring chips, which are trivial to integrate into spacecraft [26], to record the current draw of the entire flight computer as a black box [47]. Detection algorithms are then used to search for variations in current draw [47], which are often much smaller than variations due to normal operations such as cycling between power states [48]. This often leads to algorithms that emit either too many false positives or too many false negatives, affecting uptime and reliability.

Our method: utilize software-extractable features. Our insight is that instead of treating the entire compute stack as a black box, we can extract features accessible to the OS (and even in userspace) about the computer's operations to model the current draw. This can be done using system metrics, such as per-core CPU utilization, memory capacity, and memory bandwidth usage, or even more granular metrics, such as cache miss rates, extracted from the CPU's performance counters. Profiling of the program under test may also allow more insight into the current draw behavior of the CPU. Such visibility into the system will give us additional insight into program behavior that can better inform our SEL detection algorithm.

Preliminary work. Preliminary experiments suggest that incorporating system-level metrics extracted by software may significantly boost SEL detection efficacy. For example, CPU usage is highly correlated with current draw on a Raspberry Pi, a low-cost commodity single-board computer found onboard many SmallSats [26]. This correlation can be seen in Figure 1, where memory bandwidth and CPU compute was stress tested. The CPU stress test starts one

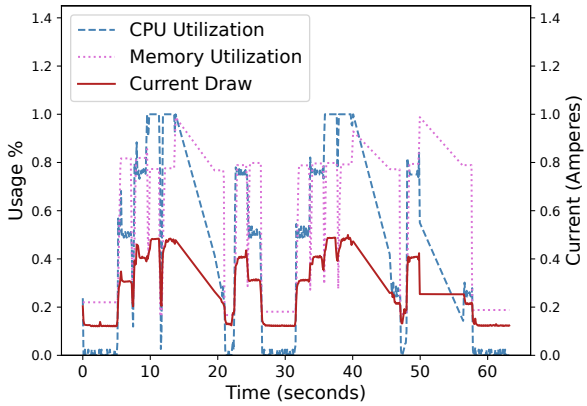

Figure 1: Example of CPU usage, memory percentage, and current draw over time under CPU and memory bandwidth stress test.

worker per core, each of which loops through a set of matrix operations, including multiplication, transposition, and addition. The memory stress test also starts one worker per core, allocates some amount of memory, and writes and reads from that portion repeatedly. As shown, the stress tests cycle between using 0, 1, 2, 3, and 4 cores on the chip, with the memory stress test cycling at an offset from the CPU stress test's usage. Across the data collected from multiple trials of similar experiments, the correlation between CPU usage and current draw was 99.9%.

We plan to collect further data both on the ground with our testbed, described in §3.2, and in flight with Cryptosat and JPL. We also collect real-world traces from our partners, as they have monitoring chips built into their spacecraft that can report the computer's current draw.

SEL detector. We present one possible approach using data collected from the testbed and the traces to detect latch-up events. This tool will run in the background of a Linux computer as a user-mode daemon and continuously record key system statistics. These statistics will be continuously tested against an algorithm such as elliptic envelope [49] that would be trained on the collected data. As seen in Figure 1, modern CPUs often record spikes in current draw, which may cause a false positive in the detection algorithm. Thus, the tool will normalize these current spikes by having the detection algorithm match against a moving window of the last 30 seconds of data. Once a suspected SEL is detected, we force a power cycle to restore the device to normal operation. We plan to explore various anomaly detection and ML algorithms and compare their efficacy and overhead.

3.2 Testing Methods

On Earth, SELs can be induced with specialized X-ray or proton beam machines [22]. CryptoSat [50] is currently using Raspberry Pi's as their flight computer and has allowed us to collect SEL data and test our mitigations in orbit. These tests are expensive and time-consuming and cannot provide a complete picture of radiation effects on software. Thus, we will need low-cost testbeds to test the effectiveness of our mitigations without the need for specialized instruments.

SEL testbed. We develop a testbed using cheap commodity parts to measure the current supplied to a Raspberry Pi board. Current data can be retrieved through the I2C interface on the Raspberry Pi. In order to simulate a latch-up, we used a USB device that can draw a variable amount of current as determined by the user. As we focus on software-based mitigations, the exact hardware configuration of a latch-up will not need to be recreated. This allows us to simulate latch-ups with varying degrees of additional current draw.

To generate data that corresponds to typical SmallSat use cases, we run various benchmarks common in scientific computing, flight software, and image and video processing. These include common operations from sklearn, FFTW, AV1, OpenJPEG, and OpenCV; and space-specific tasks from timing, location [51] and astrodynamics [52] libraries.

4 SINGLE-EVENT UPSETS

Single-event upsets (SEUs) are transient changes in a circuit's logical state caused by ionizing radiation striking the electronic device [53]. Most SEUs result in a single bit flip in memory or a spurious 1 signal traveling down a compute pipeline [44]. Anecdotally, a flight software error causing data loss on the Perseverance Mars rover has been traced to a radiation upset and has put a pause on multiple days of rover operations [30]. During regular operations, a hardened CPU on Perseverance records around one correctable SEU each Martian Sol (24.7 hours), and a non-hardened Snapdragon 801 onboard has recorded at least 4 SEUs in the past 800 Sols [30]. Simulations using a well-known radiation effects analysis toolkit [41] show that the chance of a SEU on the Snapdragon 801 is roughly 1.578×10^{-6} per bit, per day.

SEUs may affect different components of a non-hardened computer onboard a spacecraft. An SEU striking a compute pipeline can lead to incorrect data in a register [54], which can cause silent data corruption, crashes, or hangs. SEUs hitting RAM may cause similar errors when the data is processed. These errors may ultimately result in malformed packets being relayed through the network, or even corrupt transferred data in an undetectable way. As for data at rest, standard flash chips often found on spacecraft computers usually have built-in ECC which can detect and correct for SEUs. Therefore, due to the lack of any ECC mechanisms, even for single-bit errors, we assume that compute pipelines, cache, and main memory will be the primary sources of application-visible failures in a commodity computer used in space, and are thus our primary focus.

Assumptions. Since errors can affect every component of the system (even the wires!), we must assume unfortunately that not all errors can be corrected. We therefore identify *best-effort* mechanisms that can *dramatically decrease* the impact of application-visible effects caused by SEUs without inccuring a significant performance penalty.

For many local compute use cases in space, such as navigation and communication, computations can simply be redone if an error is detected [55]. Thus, we focus primarily on detection and prevention techniques rather than error correction. Observational data from Perseverance has shown only one radiation error affecting multiple bits for its entire 25-year lifespan [30]. We therefore focus on single-bit rather than multi-bit errors. Finally, despite what the title of this paper may suggest, we assume a non-adversarial fault model and focus on protection from random errors.

Due to the unique operating conditions of spacecraft, onboard computers are very constrained in their power usage and heat generation. As spacecraft batteries are limited and solar panels often do not output more power than the spacecraft draws, the power available to onboard computers are often very small [30], driving down available compute time. Furthermore, the vacuum of space does not allow computers to use convection to cool themselves, making thermal issues a major priority. Therefore, minimizing overhead, even at the cost of some reliability, may be a worthwhile compromise for spacecraft operators.

4.1 Methods

Prior methods. Early work in the SmallSat field introduced some naive solutions to mitigate SEUs. Industry's (expensive) mitigation treats the program to protect as a black box and runs it in double modular redundancy (DMR) with a hardened, dedicated ECC controller [56, 57]. As DMR duplicates every operation, it incurs at least double the runtime cost or even more if the CPU throttles due to heat.

Our approach. Our key insight is that fine-grained program behaviors, such as control and data flow, can be used to detect SEUs. We introduce *tunable DMR*, which ensures control and data flow integrity using compile-time instrumentation. These approaches can be fine-tuned to strike a balance between overhead and accuracy from just validating control flow to running in full DMR. We also introduce a software implementation of memory ECC running on underused accelerators that ensures the integrity of data stored in memory. These methods will protect data in use and in memory with significantly less overhead than current state-of-the-art methods.

Control flow integrity. Our key idea is that software programs often differ in control flow depending on whether the program completes without errors. We note that control flow within a function is determined by branch instructions, whose behavior is governed by its operand values. Therefore, in order to protect control flow, these are the critical values we will need to protect. As these values are a subset of all values in the program, we may not need to redo the entire computation to validate the correctness of control flow. Furthermore, we can create much lighter detection methods than traditional control flow integrity, as we are not expecting an adversarial fault model.

We can extract the aforementioned critical values by traversing the control flow graph (CFG) of the program and noting the values used in each transition. We can then extract the set of instructions that determine these values by traversing the use-def tree in reverse order. This list of critical values enables us to create a reference monitor that replicates only these instructions. The reference monitor can then be run in parallel with the full program or afterward to validate the control flow. If we run both the monitor and the program in parallel, we will not need to record state transitions while running the full program. However, if we run the monitor after the full program, we minimize the overhead from context switching and IPC needed when running the program.

We can modify the level of integrity provided based on mission requirements such as overhead and tolerance to errors. For example, we can choose to verify only that the transitions between basic blocks are correct rather than ensuring that the whole path is in scope. We may further improve performance by verifying transitions only between strongly connected components in the control-flow graph.

Running the reference monitor with the full program will allow us to verify that each state transition is correct while incurring less cost than running the full program twice. This method will detect any silent data corruption causing a deviation in control flow. As control flow is often dependent on the correctness of the program's data, this verification step will therefore provide a reasonable guarantee that the program executes correctly while minimizing the extra computation being done and keeping additional power draw and heat generation down.

Data flow integrity. Silent data corruption, an exceptionally challenging error to detect, may be missed by control flow integrity. As DMR is prohibitively expensive, we need a method to detect silent data corruptions that significantly alter output without doubling the overhead. Our key insight here is that not all SEUs affect the computation results equally, and only SEUs in specific bits cause large errors in the output. Therefore, if we can protect these specific bits, we may safely ignore bit flips elsewhere, as the differences in the end result will be minimal.

As a case study, we examine how to protect floating-point multiplication and division, which are expensive compared to integer or logical operations, with less overhead than

DMR. These operations are commonly used in many spacecraft workloads and can tolerate some error in the result. An SEU in a float results in relative errors up to $2^{2^{10}}$ when an exponent bit is hit, 200% if the sign bit is hit, and 50% if a mantissa bit is hit. Therefore, we can detect significant errors while minimizing overhead by protecting just the exponent and sign bits. We can further tune our method to protect a number of bits in the mantissa until we reach an acceptable margin of error.

This motivates us to apply *quantization techniques* [58] to our validation tool. Such techniques leverage the fact that most CPUs can operate faster on integers than on floating-point data to speed up calculations. In this use case, we use quantization to efficiently verify that the program output matches the order of magnitude of the ground truth. We first create a similar use-def chain as in our control flow integrity approach, but only select values that affect the final result. We convert the selected floating-point values into an integer representation of their order of magnitude and calculate the expected order of magnitude of the result based on the operations done to the values of the original program. We then validate that the orders of magnitude before and after the operations match the expected value.

Due to CPU characteristics, calculating this order of magnitude approach is faster than DMR. For example, on the ARM Cortex-A53 architecture, integer operations take up to just 2 cycles, while floating-point ones will need up to 7 cycles. Orders of magnitude can be calculated in just 1 cycle.

Coprocessor-based software ECC. In order to ensure that data stored in memory has not been affected by SEUs, spacecraft have traditionally used ECC schemes implemented in hardware [59]. However, modern low-power SoCs lack hardware error-correction codes, and running a software implementation of software ECC may be prohibitively expensive. For example, a benchmark on a Snapdragon 801 shows that verifying 2GB of memory using a software BCH coding scheme takes over 7 minutes of valuable CPU time.

We note that many of these general-purpose SoCs provide hardware accelerators to meet a variety of use cases, but they are often left unused in spacecraft. However, some of these accelerators have been shown to be capable of doing common ECC tasks [60]. Thus, we propose using these idle accelerators as software ECC, which would ensure memory integrity without visible overhead to spacecraft operators.

As a case study, we examine how the Hexagon DSP onboard Qualcomm Snapdragon SoCs can support efficient soft ECC on non-ECC memory. These SoCs act as the flight computer for many missions, including Mars rovers [29] and LEO SmallSats [27]. The DSP has access to all memory on the SoC, just as the CPU does. However, it does not have an understanding of the kernel's page table and therefore will not be able to run on pages without kernel support.

Therefore, we will pair a kernel module with a page verifier on the DSP. On startup, the kernel module will reserve an area of memory for checksums to be stored. It will then schedule pages stored in memory to checksum and pass the physical page address to the memory page verifier running on the DSP. The DSP can then retrieve the page, verify page integrity, and make repairs as needed.

Due to computational limitations on space-bound commodity computers, it is impractical and expensive to constantly cycle through the entire memory space. There may be a significant amount of time in between cycles where a SEU may sneak in undetected. Thus, we need to focus on the integrity of pages that will affect the final result of a computation. One method may be to schedule pages to be verified in least recently used order, as these pages have been in memory the longest and are thus more likely to contain an error. Another approach may involve using program traces to predict which pages will be accessed next and scheduling these pages for verification first. The verifier will also take advantage of built-in correctness checks within programs to detect errors in unscanned pages.

4.2 Testing Methods

The Perseverance Mars rover currently has a commodity computer onboard [29], which we use to collect data on the radiation environment on Mars. As data transmission to Mars is expensive and time-consuming and cannot provide a full model of the SEU rates, we also introduce alternative techniques to test programs' vulnerability to SEUs.

QEMU fault injection tool. Due to the random nature of radiation errors and the limited availability of data from space systems, we need a sufficiently accurate experimental framework that can inject errors into different components of a computer. Though architectural simulators offer better fidelity into cache and CPU pipeline behavior [61], even simple tasks like booting Linux can take minutes. This makes testing applications extremely tedious. In contrast, an instruction-level emulator like QEMU has a simpler unified memory model that boots Linux in seconds. However, as QEMU is not cycle-accurate, faults can only be injected between instructions rather than between cycles [62]. As we are primarily concerned with how errors impact computations, QEMU's granularity satisfies our needs.

We implement a QEMU simulation framework for transient radiation errors. We assume a radiation environment similar to that of normal LEO operations, with about 1 SEU per day. The framework pauses the execution of the system emulation at a selected time, and uses GDB to modify register and memory contents in the emulated system.

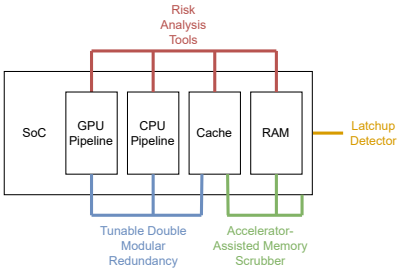

Figure 2: Diagram of processor components protected by each proposed system.

To model a cache, we use QEMU's cache plugin, which instruments memory accesses and records locations that would be stored in a cache [63]. We extend QEMU's monitor interface, which takes user input to do complex tasks such as mounting devices or taking snapshots of the virtual machine [64], to allow QEMU's cache plugin to return addresses that are located in cache or in memory.

LLVM risk analysis pass. While the QEMU-based approach provides useful error traces, it cannot trace through all execution paths and provide an upper bound on the erroneous result. In contrast, our risk analysis tool runs a control and data flow analysis of the target program and returns an error rating for each value, which quantifies the vulnerability of that value to a bit flip elsewhere.

The tool also accounts for how data is represented architecturally, producing a much more accurate error model than working with the data types at a high level. We do so by assigning a logarithmic error rating to each value representing the effect of a worst-case bit flip. For example, the maximum error of a 64-bit integer type is 2^{64}, so its error rating is 64. Similarly, the maximum error of a 64-bit float occurs when the most significant bit of the exponent is flipped, resulting in an error of 2^{1024}, so its error rating is 1024.

We then note that there are five primary mathematical operations on values that modify error ratings. The maximum error in addition and subtraction is the larger maximum error of the two operands. Thus, the error rating of such an operation is the greater of the two operands' error ratings. Similarly, the error rating of a multiplication or division operation is the sum of the two operands' error ratings. Finally, the maximum error of a modulo operation occurs when the divisor is flipped to a very large value, at which point the dividend becomes the result. Thus, the error rating of a modulo operation is the error rating of the first operand. As we are interested in worst-case error behavior, we can set the error rating of phi nodes to the largest of the incoming values.

Setting where we assign initial error ratings and what our final output value is to the endpoints of a code segment can provide us with error ratings that describe how vulnerable a piece of code is to SEUs at the basic block, strongly connected component (SCC), or function level. While such an approach does not account for error propagation in loops, this analysis is granular enough to still provide insight into error behaviors during each iteration of the loop.

5 CONCLUSIONS

Due to cost and compute requirements, LEO satellite operators are starting to use low-cost, commodity computers onboard spacecraft. These computers are susceptible to radiation errors that can damage the computer or produce incorrect results. We introduce a set of software-based methods, shown in Figure 2, that extract information about the software and hardware stack to mitigate and detect such errors. We also develop testbeds on Earth and in space to test these approaches in real-world conditions.

ACKNOWLEDGMENTS

A portion of this research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration (80NM0018D0004). This work was supported by the Department of Defense (DoD) through the National Defense Science & Engineering Graduate (NDSEG) Fellowship Program. The authors would like to thank Andrew Schmidt of the Information Sciences Institute, John Parsons of Nevis Laboratories, and Lucas Saldyt, Andrei Tumbar, Cel Skeggs, Steven Guertin, and Doug Sheldon of the Jet Propulsion Laboratory for their input on this work.

REFERENCES

[1] H. Jones. "The recent large reduction in space launch cost". In: 48th International Conference on Environmental Systems. 2018.

[2] A. C. Boley and M. Byers. "Satellite mega-constellations create risks in Low Earth Orbit, the atmosphere and on Earth". In: *Scientific Reports* 11.1 (2021), pp. 1–8.

[3] *Starlink*. 2023. URL: https://www.starlink.com/.

[4] Z. Qu et al. "LEO satellite constellation for Internet of Things". In: *IEEE access* 5 (2017), pp. 18391–18401.

[5] J. Gedmark and S. Smith. *The Evolution of the Satellite Economy*. Andreessen Horowitz. a16z Podcast, Sept. 2023. URL: https://a16z.com/podcast/the-evolution-of-the-satellite-economy.

[6] *Planet*. 2023. URL: https://www.planet.com/.

[7] *Spire*. 2023. URL: https://spire.com/.

[8] *Filecoin Foundation and Lockheed Martin Bring Decentralized Storage to Space*. May 2022. URL: https://filecoinfoundation.medium.com/filecoin-foundation-and-lockheed-martin-bring-decentralized-storage-to-space-db9a15e66264.

[9] Y. Michalevsky and Y. Winetraub. "WaC: SpaceTEE-Secure and Tamper-Proof Computing in Space using

CubeSats". In: *Proceedings of the 2017 Workshop on Attacks and Solutions in Hardware Security*. 2017, pp. 27–32.

[10] A. Fikes et al. "The Caltech space solar power project: Design, progress, and future direction". In: *Proc. IEEE WiSEE Space Sol. Power Workshop*. 2022.

[11] F. Michel et al. "A first look at starlink performance". In: *Proceedings of the 22nd ACM Internet Measurement Conference*. 2022, pp. 130–136.

[12] J. Bao et al. "OpenSAN: A software-defined satellite network architecture". In: *ACM SIGCOMM Computer Communication Review* 44.4 (2014), pp. 347–348.

[13] D. Bhattacherjee et al. "Gearing up for the 21st century space race". In: *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. 2018, pp. 113–119.

[14] S. Kassing et al. "Exploring the "Internet from space" with Hypatia". In: *Proceedings of the ACM Internet Measurement conference*. 2020, pp. 214–229.

[15] A. Singla. *Satnetlab: a call to arms for the next global internet testbed*. 2021.

[16] D. Perdices et al. "When satellite is all you have: watching the internet from 550 ms". In: *Proceedings of the 22nd ACM Internet Measurement Conference*. 2022, pp. 137–150.

[17] M. M. Kassem et al. "A browser-side view of starlink connectivity". In: *Proceedings of the 22nd ACM Internet Measurement Conference*. 2022, pp. 151–158.

[18] J. R. Wertz et al. "Methods for Achieving Dramatic Reductions in Space Mission Cost". In: *Reinventing Space Conference*. 2011, pp. 2–6.

[19] B. Yost et al. "State-of-the-art small spacecraft technology". In: (2021).

[20] L. Burcin. "Rad750 experience: The challenge of SEE hardening a high performance commercial processor". In: *Microelectronics Reliability & Qualification Workshop (MRQW)*. 2002.

[21] C. M. Schieler et al. "On-orbit demonstration of 200-Gbps laser communication downlink from the TBIRD CubeSat". In: *Free-Space Laser Communications XXXV*. Vol. 12413. SPIE. 2023, p. 1241302.

[22] S. M. Guertin. "Radiation effects on ARM devices". In: (2019).

[23] E. Birrane et al. "Linux and the spacecraft flight software environment". In: (2007).

[24] V. Verma and C. Leger. "SSim: NASA Mars rover robotics flight software simulation". In: *2019 IEEE Aerospace Conference*. IEEE. 2019, pp. 1–11.

[25] H. Leppinen. "Current use of Linux in spacecraft flight software". In: *IEEE Aerospace and Electronic Systems Magazine* 32.10 (2017), pp. 4–13.

[26] *Private correspondence with LEO SmallSat operator*. 2022.

[27] J. Bosch-Lluis et al. "The Smart Ice cloud sensing (Smices) SmallSat Concept". In: (2020).

[28] D. Selva and D. Krejci. "A survey and assessment of the capabilities of Cubesats for Earth observation". In: *Acta Astronautica* 74 (2012), pp. 50–68.

[29] B. Balaram et al. "Mars helicopter technology demonstrator". In: *2018 AIAA Atmospheric Flight Mechanics Conference*. 2018, p. 0023.

[30] *Private correspondence with space agency*. 2022.

[31] S. Ghemawat, H. Gobioff, and S.-T. Leung. "The Google file system". In: *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 2003, pp. 29–43.

[32] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[33] V. Nwankwo, N. N. Jibiri, and M. T. Kio. "The impact of space radiation environment on satellites operation in near-Earth space". In: *Satellites Missions and Technologies for Geosciences* (2020).

[34] S. Ma et al. "Network Characteristics of LEO Satellite Constellations: A Starlink-Based Measurement from End Users". In: *arXiv preprint arXiv:2212.13697* (2022).

[35] S. A. Jyothi. "Solar superstorms: planning for an internet apocalypse". In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 2021, pp. 692–704.

[36] G. S. Rodrigues et al. "Analyzing the impact of fault-tolerance methods in ARM processors under soft errors running Linux and parallelization APIs". In: *IEEE Transactions on Nuclear Science* 64.8 (2017), pp. 2196–2203.

[37] Y. Li et al. "A case for stateless mobile core network functions in space". In: *Proceedings of the ACM SIGCOMM 2022 Conference*. 2022, pp. 298–313.

[38] W. Schimmerling. "The space radiation environment: an introduction". In: *The Health Risks of Extraterrestrial Environments* (2011).

[39] A. A. Hwang, I. A. Stefanovici, and B. Schroeder. "Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design". In: *ACM SIGPLAN Notices* 47.4 (2012), pp. 111–122.

[40] S. M. Guertin et al. "Recent SEE results for Snapdragon processors". In: *2019 IEEE Radiation Effects Data Workshop*. IEEE. 2019, pp. 1–5.

[41] B. D. Sierawski et al. "CRÈME-MC: A physics-based single event effects tool". In: *IEEE Nuclear Science Symposuim & Medical Imaging Conference*. IEEE. 2010, pp. 1258–1261.

[42] A. J. Tylka et al. "CREME96: A revision of the cosmic ray effects on micro-electronics code". In: *IEEE Transactions on Nuclear Science* 44.6 (1997), pp. 2150–2160.

[43] R. Ecoffet. "Overview of In-Orbit Radiation Induced Spacecraft Anomalies". In: *IEEE Transactions on Nuclear Science* 60.3 (2013), pp. 1791–1815. DOI: 10.1109/TNS.2013.2262002.

[44] J. A. Pellish. *Radiation 101: Effects on Hardware and Robotic Systems*. Tech. rep. 2015.

[45] D. M. Hassler et al. "The radiation assessment detector (RAD) investigation". In: *Space science reviews* 170.1 (2012), pp. 503–558.

[46] K. L. Bedingfield and R. D. Leach. *Spacecraft system failures and anomalies attributed to the natural space environment*. Vol. 1390. National Aeronautics and Space Administration, Marshall Space Flight Center, 1996.

[47] A. Dorise et al. "Machine learning as an alternative to thresholding for space radiation high current event detection". In: *2021 21th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*. IEEE. 2021, pp. 1–7.

[48] Y. Wang et al. "Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86". In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 679–697.

[49] P. J. Rousseeuw. "Least median of squares regression". In: *Journal of the American statistical association* 79.388 (1984), pp. 871–880.

[50] *Crypto-satellites that power blockchain and cryptography*. 2023. URL: https://www.cryptosat.io/.

[51] C. H. Acton Jr. "Ancillary data services of NASA's navigation and ancillary information facility". In: *Planetary and Space Science* 44.1 (1996), pp. 65–70.

[52] D. E. Gaylor, T. Berthold, and N. Takada. "Java Astrodynamics Toolkit (JAT)". In: *Advances in the Astronautical Sciences* 121 (2005), pp. 263–272.

[53] E. Normand. "Single-event effects in avionics". In: *IEEE Transactions on nuclear science* 43.2 (1996), pp. 461–474.

[54] N. J. Wang, J. Quek, T. M. Rafacz, et al. "Characterizing the effects of transient faults on a high-performance processor pipeline". In: *International Conference on Dependable Systems and Networks, 2004*. IEEE Computer Society. 2004, pp. 61–61.

[55] W. M. Owen Jr. "Methods of optical navigation". In: (2011).

[56] A. G. Schmidt, M. French, and T. Flatley. "Radiation hardening by software techniques on FPGAs: Flight experiment evaluation and results". In: *2017 IEEE Aerospace Conference*. IEEE. 2017, pp. 1–8.

[57] C. A. Skeggs. "Vivid: An Operating System Kernel for Radiation-Tolerant Flight Control Software". MA thesis. Massachusetts Institute of Technology, 2022.

[58] C. Zhu et al. "Trained ternary quantization". In: *arXiv preprint arXiv:1612.01064* (2016).

[59] M. Reid and G. Ottman. "Software controlled memory scrubbing for the Van Allen probes solid state recorder (SSR) memory". In: *2014 IEEE Aerospace Conference*. IEEE. 2014, pp. 1–6.

[60] S. Keskin and T. Kocak. "GPU accelerated gigabit level BCH and LDPC concatenated coding system". In: *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2017, pp. 1–4.

[61] J. Lowe-Power et al. "The gem5 simulator: Version 20.0+". In: *arXiv preprint arXiv:2007.03152* (2020).

[62] M. Kaliorakis et al. "Differential fault injection on microarchitectural simulators". In: *2015 IEEE International Symposium on Workload Characterization*. IEEE. 2015, pp. 172–182.

[63] *QEMU TCG Plugins*. 2023. URL: https://www.qemu.org/docs/master/devel/tcg-plugins.html.

[64] *QEMU Monitor*. 2023. URL: https://www.qemu.org/docs/master/system/monitor.html.