Linux Boot Failures Under Proton Irradiation

Aditya Bhattacharya*, Gelmar Luiz da Costa*, Rafał Graczyk‡,
Jan Swakoń†, Leszek Grzanka†, Sebastian Kusyk†, Marcus Völp*
*Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg
aditya.bhattacharya@uni.lu, gelmar.costa@uni.lu, marcus.voelp@uni.lu
†Department of Radiation Research and Proton Radiotherapy, Narodowe Centrum Badan Jądrowych (NCBJ), Poland
jan.swakon@ifj.edu.pl, leszek.grzanka@ifj.edu.pl, sebastian.kusyk@ifj.edu.pl
‡Independent Researcher
rafal@graczyk.io

***Abstract*—To meet the ever-growing computational demand of high-performance applications, and specifically to keep up with the trend of AI and increased autonomy, space-based systems must be equipped with and operate increasingly more powerful computing systems. At the same time, costs demand using commercial off the shelf (COTS) systems and the functionally rich operating systems they run. However, naturally, these systems have not been built for the harsh environment of space, and even in low-earth orbit, such systems are exposed to radiation levels, which may cause software to fail and, when left untreated, may crash the entire system permanently. Rejuvenation is one method for addressing radiation faults. It involves restarting and power cycling the computer system and rebooting its software stack to return it to a state in which it can continue to operate correctly. However, when applied to COTS systems, frequent reboots become necessary, and while the radiation effects of software running on radiation-hardened systems is relatively well understood, the same is not true for the boot process of legacy operating systems, such as Linux, on COTS platforms in radiative environments. To bridge this gap, we performed an up to 58 MeV proton radiation test, observing two Linux-based operating systems booting on an iMX8 Plus compute module and report here on our results. We found Linux to be significantly less resilient to radiation faults until it has properly configured the processor's fault handling capabilities. Moreover, while file system failures were the most frequent, failures in the Linux core and driver initialization code caused most of the crashes.**

***Index Terms*—Boot, Operating System, Radiation, Rejuvenation**

I. Introduction

Radiation induced faults are a major threat to computer systems deployed in space. However, there are also areas on ground where systems have to withstand high levels of radiation. Examples include medical equipment, in particular in radiology, robots for cleaning up sites such as Chornobyl, Fukushima, and a few other areas. A significant amount of research has already focused on understanding radiation effects on electronics [1][2][3][4] and mitigating radiation-induced effects [5][6][7]. However, these techniques often require specialized hardware, hardened against radiation, which translates to high costs when designing and taping-out systems for this specific purpose.

Commercial-off-the-shelf (COTS) systems, on the other hand, are produced for mass markets and the performance requirements they entail. In particular, mobile-phone SoCs are produced in millions, amortizing design and tape-out costs, while delivering a huge performance gain, including for AI, when compared to classical space-grade equipment.

Of course, because they are not built for space, COTS computing systems cannot withstand the effects radiation induces in space on their own. High-energy charged particles, such as protons, may induce so-called single event effects (SEEs) as they strike through semiconductor components, leading to transient disruptions, such as single-event upsets, which manifest, for example, as bit flips in memory, in the processors' logic, or in processor registers. When read by subsequently executing parts of the programs and system software that the processor runs, such upsets may propagate, causing additional failures at the software level and, ultimately, software crashes, wrong control signals, and the like. Desired functionality will no longer be provided, which is why failures of this kind are also summarized as single-event functional interrupts (SEFI). Resetting software (e.g., by reloading the state from protected memory) is an effective countermeasure to SEFIs.

Single-event latch-up (SEL) is another such single-event phenomenon, occurring when there is a sudden build-up of charge, e.g., from an ion strike; this extra charge may turn on the parasitic thyristor inherent to most CMOS processes. When an SEL occurs, it can short-circuit some internal power lines, which is observed as a current surge or as more gradual current increments depending on the complexity and topology of the IC. A surge in current flow would result in increased local thermal dissipation and, in the worst-case scenario, can cause permanent damage to the circuit. SELs can be removed by completely stopping power flow through the thyristor, i.e., by shutting off the power supply to the device, while ensuring the device will also not be getting power supply from any of the I/O pins.

Ultimately, total dose effects permanently damage circuits.

This work is supported by the European Commission through the Horizon 2020 projects RADNEXT and Eurolabs, as well as by the Fond Nationale de Recherche in Luxembourg through the FNR-Core project HERA (CS20/IS/14689454). In fulfillment of the obligations arising from the grant agreement, the author has applied a Creative Commons Attribution 4.0 International (CC BY 4.0) license to any Author Accepted Manuscript version arising from this submission.

At least for memory, the system's BIOS and its legacy OSs are prepared to compensate permanent memory faults during reboot by avoiding corrupt locations after the built-in self-test.

All the above requires rebooting the system and, in case of COTS systems, much more frequently than before, including during critical parts of a mission. In recent years, much work has been done to understand and categorize radiation-induced faults and investigate mitigation techniques of COTS complex systems. However, the fault behavior of the boot process of complex operating systems under radiation is less well understood, and there is almost no literature on investigating the same. We have observed that these devices extensively fail to boot in radiative environments. Our goal is to better understand the OS boot fault mechanisms and to work towards developing mitigation techniques. To that end, we carried out a radiation test campaign in the cyclotron facilities of the Department of Radiation Research and Proton Radiotherapy at The Henryk Niewodniczański Institute of Nuclear Physics of the Polish Academy of Sciences (IFJ PAN). We investigate the boot process of two Linux-based systems under up to 58 MeV proton radiation, booting on a iMX8 Plus COTS compute module; after our previous radiation test revealed that this computer module is the least vulnerable, compared to a Raspberry Pi Zero [8] or an Orange Crab FPGA [9].

Macroscopically, we found that the Linux kernel is significantly more vulnerable while it is still booting but then enters a phase of relatively more resilience to radiation-induced faults once all the processor's and operating system's fault-handling capabilities are properly configured. In this paper, we provide a justified explanation of our experimental setup, a detailed analysis of our findings, characterize observed faults, carry out a root cause analysis, and attribute the faults to specific stages of the boot process. After providing the necessary background and relating our work to the works of others, we describe our radiation experiments in Section III, present and analyze the results in Section IV, and discuss our findings in Section V. Section VI draws conclusions and hints at directions for future work.

II. Related Work and Background

Radiation testing of COTS components has been of interest to the scientific community for some time now, and as such, a significant amount of work has been done to establish the framework and methodology for radiation testing, as well as understanding the susceptibility of different COTS hardware under radiation. Previous works by Guertin et al.[10] have tested the susceptibility of the Verdin imx8m hardware to irradiation. The works of Frias-Dominguez et al. [11] analyze the reliability of system reboots under irradiation, and the works of Miller et al. [12] have focused on an implementation of Linux on LEO-based commercial hardware. However, understanding the error mechanisms of Linux boot failures under radiation is a gap that we have come across often and is what we aim to bridge with this article.

A. COTS systems and Legacy OSs for space

Normally, operating systems are expected to boot once and then remain active for the duration of the segment of a mission. More precisely, they are typically booted in situations where the mission is not yet critical and where boot failures may be easily compensated by retrying the process several times until the boot concludes successfully. This is even true for classical old-space systems when activating a redundant shuttle computer after the primary system has failed, which was required on several occasions in the past[13]. However, when transitioning from radiation-hardened systems to commercial-off-the-shelf (COTS) platforms and from specialized operating systems, such as the widely popular choices of the space community RTEMS[1], and VxWorks[2], to legacy monolithic operating systems, such as Linux[3].

Rich legacy operating systems, such as Linux, become increasingly relevant in radiation-rich environments since they offer the functionality that applications and specifically the application-level components of autonomous space vehicle control require. For instance, many artificial intelligence (AI) applications and similar analyses require access and multiplexing of GPU or TPU accelerators, which require drivers, libraries, and sometimes even additional user-level components to be operated. Although embedded versions for bare-metal operation exist for some of those, their performance is often limited, and the additional benefit of a rich OS with its dynamic resource allocation and scheduling capabilities comes in handy to cope with this increased complexity. Typical examples of such workloads include AI-based analysis of payload data to identify interesting aspects in the data before compressing and communicating samples to the ground. Increased mission autonomy also requires complex AI-based stacks to remain operational, sometimes even during critical situations where repeating the boot process until the platform is up and running is not an option.

B. Understanding how Linux boots

The boot process of legacy operating systems is significantly longer, as they follow a monolithic structure and have to initialize more subsystems before the fault-handling capabilities are fully configured. In the case of the Linux kernel, which is a monolithic kernel, essential kernel services and modules such as memory management, process scheduling, filesystem, networking, etc., must be initialized before the user-space handoff; however, this does not imply that all kernel services need to be fully operational before the user applications start. The user applications can be started as early as the essential kernel core modules are initialized. However, additional kernel services may still need to be initialized before the applications that depend on them can work. After the essential kernel services are initialized, the system continues

[1]https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Software_Systems_Engineering/RTEMS

[2]https://www.windriver.com/products/vxworks

[3]https://kernel.org

to bring up additional kernel services based on dependencies and parallelize them when possible.

This and the fact that COTS platforms, not being built for the harsh radioactive environment one encounters in space, are more susceptible to radiation-induced faults, making them more likely to experience faults during the boot process. Power cycling the cores of a platform running support rich operating systems and doing so also during critical situations significantly increases the number of times when during a mission the legacy operating system has to boot, and also adds the requirement that the boot process should complete successfully, at least most of the time, since otherwise, the mean-time-to-recovery may easily exceed the mean time between failures.

During the boot process, a device spends most of its time in kernel mode and is highly susceptible to failure. Although much work focuses on radiation-induced runtime failures in such systems, only a few focus on protecting the system during boot.

But why is the boot process relevant? Most radiation-induced faults are transient, such as bit-flips in memory, in the processor registers or its logic, with typically little to no effect on the correct or approximately correct execution of applications. Bit-flips are benign when the memory location is overwritten before the value is read. Logic faults can be tolerated if computations are only slightly off and corrected during the next control cycle. Latch-ups and static surges can only be reversed by power cycling the system, of which rebooting is an unavoidable part. Also, when dealing with complex systems, they fail in complex ways; and often it becomes necessarily obvious that the OS has become unstable due to some faults that occurred, or even certain functionality may begin to fail, e.g. docker images may shut down and applications may not respond. Although many of the faults present with the same signature to the user at the higher level, it is impossible to pinpoint the lower-level root cause fault and mitigate it. In cases where complex systems are dealt with, rebooting the system is a standard method of addressing such symptoms before they cascade into bigger failures.

C. Shielding, Hardening, and Error Correction

To understand why COTS systems require special protection mechanisms, we first have to review the standard mechanisms for protecting special purpose space compute systems from radiation.

Applicable to both space-grade and COTS hardware is the protection from ionizing and non-ionizing radiation through shielding. Shielding adds extra material between the paths of radiation and the electronic components to prevent the former from hitting the latter or to influence their energy profile when they do so. Shielding naturally adds extra weight and thus increases launch costs. However, unlike the other methods, it is also viable to reduce total dose effects. A second negative aspect of shielding is secondary radiation due to high-energy particles hitting the matter in the shield. Negative effects of this sort are typically minimized by selecting appropriate materials (e.g., the ISS leverages its water supply to shield astronauts and other equipment [14]).

Some materials from which electronics can be built are immune to certain classes of radiation effects. For example, silicon on insulators will not be susceptible to latchups in the same way that other processes are [15]. Although compatible with COTS design in the sense that custom chips can be created given access to the COTS system description (e.g., at register transfer level or layout), the costs of taping out custom hardware can be significant if not compensated by mass market production.

This leaves error correcting codes (ECCs) and their implementation in register, computation, and memory logic [16][17]. For example, space-hardened designs, such as LEON leverage ECCs throughout several pipeline stages, including in the register file and for critical computations. The latter can as well be protected by means of encoded processing or by executing computations in lock step, while comparing the results. The former thereby performs functionally identical but different computations, such as instead of adding a+ba + b, performing the operation (ca+cb)/c(ca + cb)/c.

Typically ECC is configured to detect two and correct one fault in a data word, with other configurations being available. Also correction bits are typically stored separately to not subject them to the same proximity faults.

While including ECC memory is relatively straight forward for COTS systems, this comes at the costs of an additional performance penalty, both in terms of the latency for en- and decoding the written value and, more significantly, for scrubbing ECC memory cells to reset their correction ability after bits have flipped. Scrubbing periodically passes through all ECC protected cells, reads their value and writes them back, which re-encodes the correctable bitflips and resets the ECCs correction capacity.

In our experiments, the DUT has inline ECC-enabled DDR4 RAM. However, the ECC logic is initialized during the user-space handoff after the kernel boot phase is completed. Thus, the RAM does not have ECC active during the kernel boot phase.

D. Rejuvenation as Fault-Mitigation Strategy

In this paper, we analyze the Linux boot process under radiation to evaluate the possibility of using an alternative radiation mitigation strategy, namely to proactively or reactively (after failures have been detected) rejuvenate the system, by removing faults and returning the system to a known good state.

Rejuvenation was first proposed by Sousa et al. [18] in the context of creating continuous fault tolerance and resilience to cyberattacks. It involves removing adversarial code, and, in case it is used for attack tolerance, diversification of software (e.g., by means of address space layout optimization) and may imply that a patched version is loaded after reset.

Techniques may further recover application downtimes by migrating application state to pre-launched instances, which then take over. Matovic et al. [19] has shown that it is also

possible to rejuvenate fine granular control tasks, even during time-critical operations.

For COTS systems, rejuvenation entails rebooting and power cycling the entire module, algorith as part of our future work, we will also investigate more fine granular options.

III. Irradiation Experiments

In the following, we provide details on our experimental setup for our irradiation experiments. The selection of the NXP iMX8M compute module originates from a previous radiation test, involving several co-authors of this work, where we compared the performance of a Raspberry Pi Zero 2, an Orange Crab FPGA and the iMX8M board under load. The latter performed best throughout all radiation levels. We observed that for most flux and energy levels, the boot did not succeed, which motivated us to conduct the present test.

For example, at 58 MeV only 50% / 30% and 1% of the boot attempts succeeded at a flux of 3.0E+06, 7.0E+06, and 1.0E+07, respectively. No successful boot could be observed between 50 MeV and 58 MeV when increasing the flux. For our further investigation, we have therefore chosen to focus on the critical energy levels and flux settings. For the high range, we have selected an energy level in the middle of these two values, namely at 54 MeV.

Similarly, at a constant flux of 1.0E+7, only 20% of all boot attempts succeeded for lower energy rates, with an outlier at 20 MeV where no boot was successful. We associate this outlier with slow protons having a higher chance of interacting with memory elements to cause unrecoverable bit flips. Research shows that a distribution of $>\sim$2.5 MeV electrons and $>\sim$29 MeV protons in SAA have been determined over 27 years[20], as such 20 MeV is a good representation of the energy levels experienced by devices in orbit.

It should be noted that the idea behind a radiation test is not to replicate a radioactive environment such as outer space but to conduct accelerated tests on the device under test to understand its susceptibility to radiation. The proton fluxes used in the experiments are more than one order of magnitude higher than the absolute worst case situation one can find on LEO[21].

A. Beam Profile and Configuration of the Radiation Facility

To characterize the fault behavior of a legacy operating system booting, we subjected two Linux based operating systems, running on an NXP iMX8M compute module, to proton radiation at 20 MeV and at 54 MeV. We performed the tests at the AIC-144 cyclotron facilities of the Henryk Niewodniczański Institute of Nuclear Physics of the Polish Academy of Sciences (IFJ PAN). The beam was configured to 40mm diameter and focused on the SoC of the NXP iMX8M module.

Figure 1 shows the 2D beam profile that we captured during our experiments. The center of the 2D profile represents the actual targeted diameter of the beam in the dark red region, along with the scattering observed on the two lower quadrants of the profile. Aside from on-chip caches, AI and crypto

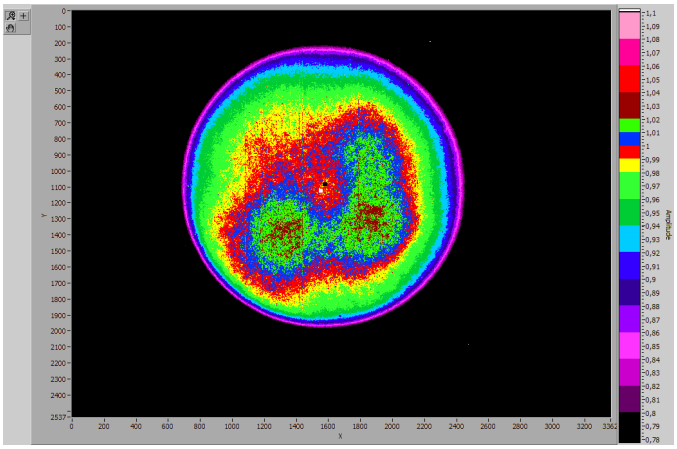

Fig. 1. 2D beam profile during our experiments

accelerators, and an ARM Coretex M7 core running at 800 MHz. The SoC contains four ARM Cortex A53 cores running at 1.6 GHz, on which the two Linux operating systems were booted. We frequently rebooted our device under test (DUT) whenever we observed it crashing or after a successful boot. Overall, we observed Linux booting under radiation for 90 minutes, during which time we observed 11 boots for each of the two OS configurations, with 22 attempts to boot in total.

B. DUT Configuration and Experimental Setup

We configured our device under test with two different Linux distributions to focus on the effects happening in the kernel while ruling out most influences from a particular distribution. The first is a minimal embedded Linux, built with the custom Toradex Board Support Package (BSP) for the iMX8M. This configuration is provided by the board manufacturer Toradex. The second is TorizonOS, which builds on top of the Toradex Linux BSP based distro by adding features such as containerization, over-the-air updates, device monitoring, etc. Both distributions run on the same Linux kernel version 5.15. The memory footprint of TorizonOS is 205 MB bigger than that of the Toradex Linux BSP (445 MB vs. 240 MB in total). As an application workload, we installed an algorithm for attitude determination and control (ADCS). We consider a boot to be successful if it enters the runtime environment and starts the ADCS application.

We placed the DUT in the proton beam and while the latter was active, we attempted to boot the device. In case the boot was successful, we initiated a reboot in software using the Linux command `shutdown -r`, which properly shuts down the operating system, writes back files, resets the cores, and reboots the system, but does not power-cycle the DUT. Also, because shutdown ultimately manifests itself in a soft reset of the core, the device will not perform a power on self-test (POST) during startup. Once we observed a boot attempt to fail, we initiated power cycling by hand, a process that would be automated in deployed systems. We do so by switching off the power supply to the device to cause a hard reset. After turning the power supply on, the system considers an

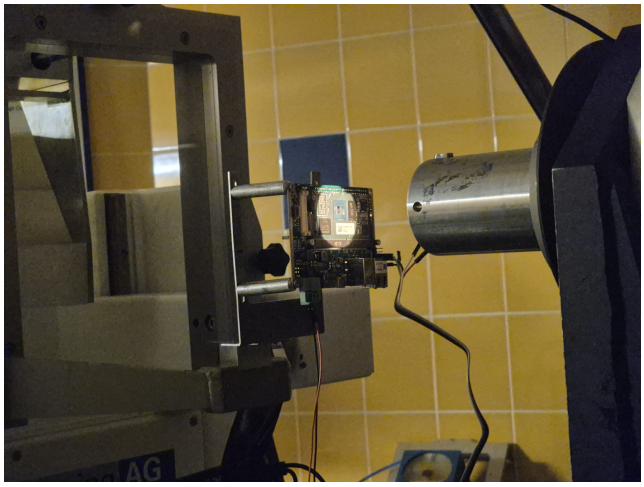

Fig. 2. Device under proton beam in test chamber

unexpected power off and attempts to restore the file system consistency by replaying the older journal entries. Of course, this does not succeed because the journal entries are corrupt, and it clears the file system journals and metadata to boot from a pristine condition; however, this takes longer due to the retained journals being replayed at the start.

In both cases, the system assumes memory to be completely cleared and will re-allocate, initialize, and configure all data structures it needs (e.g., interrupt vector tables, page tables, etc.) during the boot and while running.

We repeated the above tests until the DUT experienced an unrecoverable failure, leaving the device unresponsive.

To place the device under test in the proton beam, we mounted it on a Verdin Mallow carrier board, which we connected via its UART and a UART to USB bridge to an external PC that we used for capturing the logs and initiating the software reboot. We configured the kernel to dump its bootlogs on the serial port and configured the port to the highest available baud rate of 115200. The logs were captured on an external PC using a serial monitor and saved for additional redundancy in the flash memory of the device. We found no discrepancy between stored and communicated kernel logs. Figure 2 shows the complete setup with the cables dragging down from the highlighted DUT in the middle being the power supply and serial line cables.

We scaled the beam profile from Figure 1 according to the dimensions of the DUT, along with centering the beam profile with the center of the beam's region of influence from Figure 2, to create a superimposed illustration of the beam profile on the DUT in Figure 3.

The beam was aimed at the NXP Imx8mp processor on the SoC. However, we made sure that the flash memory on the left of the processor and RAM on the right are also affected by the beam's area of influence and are irradiated significantly. This becomes interesting when observing the failures, as most originate from SEUs occurring in the RAM and the flash memory, reported by its controller. After the test

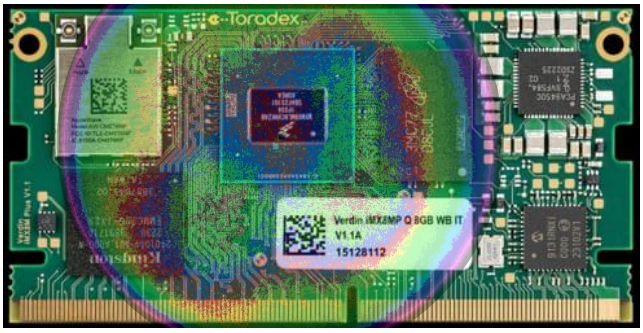

Fig. 3. Superimposition of beam profile over DUT

campaigns, we evaluated the captured logs to identify failures and understand the error mechanisms.

IV. Results and Analysis

In the following, we report on our observations and findings from retrieving the kernel logs during bootups. We report the variations in power consumption for different boot methods and successful versus failed boot attempts. We make a consolidated list of boot failure symptoms by going through the boot logs and removing expected errors (e.g., missing display, which was not connected) and repeated errors. Using the identified symptoms, we carry out a root cause analysis to investigate the cause and effect of the faults. We finally attribute the faults to specific stages of the boot process, identifying particularly vulnerable components and stages.

A. Power Consumption

The DUT was powered with an external variable power supply, and the power consumption of the device was continuously measured. We consider the total power consumption of the DUT for each boot attempt by sampling the power supply current every second and plotting the instantaneous power curves for each boot attempt.

We then assign to each power consumption curve the type of boot we observed to observe discrepancies in the power consumption variations for different situations. Here, our findings:

- **Successful Boot (Cold Start):** The power consumption exhibits multiple peaks, with the highest reaching approximately 2.8-3.0W before stabilizing. The transitions are relatively smooth, and the power settles into a steady operating range after initialization.
  Boot Health Indicator: A well-defined peak followed by stable power consumption indicates a healthy boot. Any deviation, such as excessive fluctuations or prolonged high power, may suggest boot delays or hardware initialization issues.
- **Successful Boot (Soft Reboot):** The power drawn remains moderate, typically within 1.5-2.0W, with smoother transitions compared to a cold start. The peaks are lower and less frequent, indicating a more efficient handoff between reboot phases.

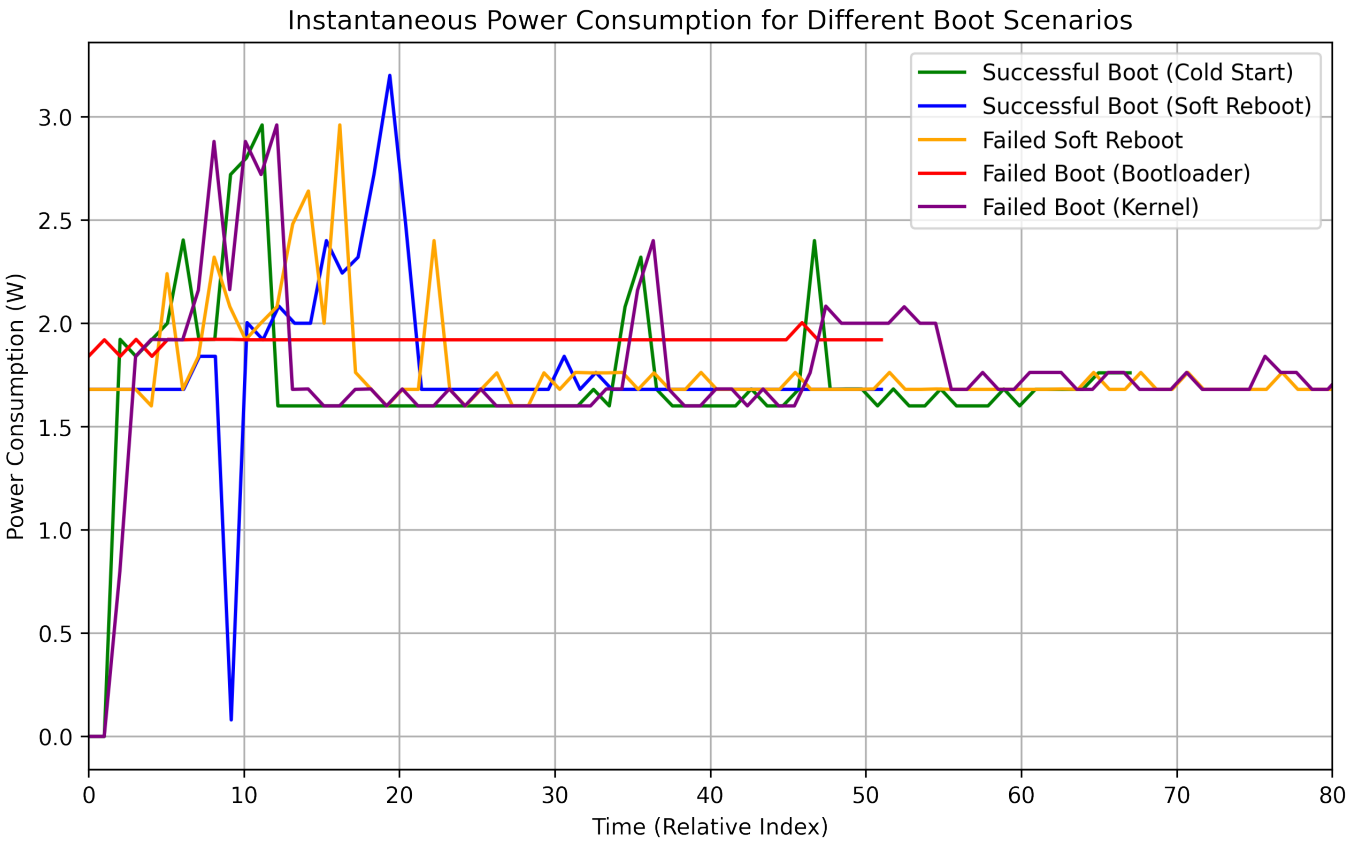

Fig. 4. Power consumption curves for different boot scenarios

Boot Health Indicator: A lower power range and minimal fluctuations characterize a healthy soft reboot. Inconsistencies in this pattern may indicate resource leaks, firmware inefficiencies, or improper shutdown states prior to reboot. In particular it can be seen that some boot phases are skipped during software reset.

- **Failed Soft Reboot:** The power profile follows a similar trend to a soft reboot but becomes erratic before failing to stabilize. A final peak before plateauing suggests an unsuccessful transition to user space.
  Boot Health Indicator: If power remains unstable or fluctuates without reaching the expected steady-state, the system is likely encountering a kernel panic, driver failure, or file system corruption during reboot.
- **Failed Boot (Bootloader Issue):** The initial power ramp-up is present but does not reach the expected peak levels. Instead of stabilizing, the power fluctuates inconsistently, potentially indicating repeated bootloader retries.
  Boot Health Indicator: A failure to reach peak power followed by repeated fluctuations suggests the bootloader is stuck in a loop. This could be due to missing boot configurations, corrupted firmware, or a hardware fault.
- **Failed Boot (Kernel Panic or Filesystem Issue):** The power usage increases in an expected manner but fails to stabilize. In some cases, sharp fluctuations precede a power drop-off, suggesting system failure during the kernel initialization phase.
  Boot Health Indicator: If power rises but never stabilizes, the kernel may be encountering a fatal error. Identifying whether the failure occurs at a consistent power threshold could help pinpoint whether the issue is kernel-related or due to peripheral initialization failures.

B. *Symptoms identified in bootlogs*

Our study of the boot log indicated the following specific errors, which ultimately resulted in unsuccessful boot attempts. We report the timestamp in the kernel log, relative to the start of the boot, the error message, and a more elaborate explaination. Also, some errors trigger recovery mechanisms in which the kernel already attempts to resolve the error at hands. We list them as subsequent errors.

- **Cache Flush Errors:**
  `[25.189036] mmc2: cache flush error -110`
  Several `cache flush error -110` messages indicate problems when attempting to write cached data to the flash storage device. This is often associated with device or hardware communication issues to the eMMC device, They indicate single-event upsets in the embedded memory controller for the eMMC.

- **EXT4 Filesystem Errors:**
  `[39.561741] JBD2: Error -5 detected when updating journal superblock for mmcblk2p1-8.`
  `[41.738854] EXT4-fs error (device mmcblk2p1): ext4_journal_check_start:83: comm systemd: Detected aborted journal`
  These errors indicate that the EXT4 filesystem journal detected corruption. Journals are used to maintain filesystem integrity, and corruption here points to underlying storage issues, typically single-event upsets in the file system journal.
- **I/O Errors on the Storage Device (`mmcblk2`):**
  `[41.685291] blk_update_request: I/O error, dev mmcblk2, sector 8192 op 0x1:(WRITE) flags 0x23800 phys_seg 1 prio class 0`
  These errors indicate that read/write operations to the `mmcblk2` device (the flash storage containing the OS image) are failing. Multiple sectors and operations (read/write) are affected, trying to access invalid memory locations. This points to a failure of the primitive file system, for example, in case the OS image remained corrupted upon the reboot attempt. We were able to trace back this issue to `sdhci_send_command` warnings and `sdhci_request` functions failing in the `sdhci.c` driver, of course, this again points to a failure of the memory controller.
- **Buffer I/O Errors:**
  `[41.696197] Buffer I/O error on dev mmcblk2p1, logical block 0, lost sync page write`
  These errors suggest that the system is unable to write data to the buffer for the specified block device (`mmcblk2p1`), which corresponds to a partition on the flash storage.
- **Failure to Read Inodes and Directory Blocks:**
  `[43.535470] EXT4-fs error (device mmcblk2p1): __ext4_find_entry:1678: inode #1701318: comm systemd: reading directory lblock 0`
  These errors indicate that the filesystem cannot read directory entries or inodes. This likely occurred because of corruption caused by failed writes.
- **Subsequent errors:**
  `[41.774248] EXT4-fs (mmcblk2p1): Remounting filesystem read-only`
  Due to persistent I/O errors, the filesystem is remounted as read-only to prevent further damage.
  Such errors also present with varying logs on the LinuxBSP based DUT
  `[49.244343] JBD2: Error -5 detected when updating journal superblock for mmcblk2p1-8.`
  `[66.733880] EXT4-fs (mmcblk2p1): I/O error while writing superblock`
  `[66.740182] EXT4-fs (mmcblk2p1): Remounting filesystem read-only`
- **Service Failures:**
  Critical services fail to start:
  `Failed to start greenboot MotD Generator.`
  `Failed to start container runtime (e.g., Docker).`

C. Root Cause Analysis and Failure Cascades

In the following, we return to the above errors, analyzing their root causes and how failures cascade.

1) blk_update_request: I/O error: This error indicates that the storage region is inaccessible. The root cause was a flash memory misconfiguration due to incorrect memory mappings, which prevented proper read/write operations to the storage device. The memory controller of the flash device failed due to errors occurring while executing the code in the sdhci.c file, indicating that the file system does not find the correct memory addresses in the flash memory space. This can occur from a corrupt pointer value or register locations. While the data integrity of the flash storage remained intact, corrupt registers in the memory controller cause the filesystem to not communicate properly with the flash storage. Loss of read or write capabilities to the eMMC has led to further corruption of the filesystem, and driver initialization failures.

- **Memory Issue Type:** Storage region inaccessible
- **Root Cause:** flash memory misconfiguration due to bad mappings

2) EXT4-fs error: ext4_find_entry: The error is caused by filesystem metadata being located in non-existent memory. This typically results from storage failures or uninitialized memory regions, leading to an inability to locate file system entries.

- **Memory Issue Type:** Filesystem metadata in non-existent memory
- **Root Cause:** Storage failure, uninitialized memory

3) JBD2: Error updating journal superblock: This issue arises due to missing journal metadata. The root cause is corrupt memory allocation for journaling, which prevents the journal from properly recording filesystem changes, potentially leading to data loss.

- **Memory Issue Type:** Journal metadata missing
- **Root Cause:** Corrupt memory allocation for journaling

4) systemd: Failed to start service: This error is caused by userspace memory corruption. The root cause is missing dependencies due to earlier faults in the boot process, preventing essential system services from starting successfully.

- **Memory Issue Type:** Userspace memory corruption
- **Root Cause:** Dependencies missing due to earlier faults

Exploration of the faults and searching for their root causes led us to classify the faults into two categories: root cause failures, and cascaded failures. The root cause failures occur

deeper in the system and during the earlier stages of the boot process. They often go undetected. However, it is likely for them to cause cascaded failures as a consequence of functionalities not properly initialized or the attempt of dependent functions trying to use failed functionalities.

Figure 5 illustrates the mechanism of cascading failures. Levels 0 to 4 denote the stages in which the failures occur. The blocks in a lighter blue gradient denote failures that are not observable and do not appear on any error reporting system during the boot, and have been identified during our root cause search. The blocks in solid blue color denotes failures that are observable and can be seen as error messages.

D. Correlating Failures to the Boot Stages

The complete boot process occurs in 4 distinct phases; bootROM, bootloader, kernel boot, userspace init. The kernel boot phase is the longest of these phases, in a scenario where the user space does not need to initialize many applications during the user space handover. This stage is also the most vulnerable because it is the most primitive in the sense that the kernel's fault handling capabilities are only installed in this stage and are thus not available to recover from faults in this stage. The kernel core and necessary peripheral drivers are initialized in this phase, and most of the repair and security services are not initialized until the user space handover. We observed the majority of errors occurring in the kernel boot phase and, as such, identified the kernel as the most vulnerable component during boot. Here is our explanation of the stages of the kernel boot phase and how errors from one stage cause cascading errors in the following stages.

1) Kernel Core Initialization: First, the kernel image is decompressed and relocated to memory, after which it begins parsing the device tree (DTB) to understand the hardware configurations. An early console is initialized to enable debugging output, while fundamental system components such as memory, CPU cores, and basic clocks are brought online. At this stage, we see that the console is working as errors such as "`imx-drm display-subsystem: no available port`" indicate that the Direct Rendering Manager (DRM) subsystem fails to detect an available display port, and this is expected as the DUT does not have a display device. We see no interesting errors occurring in this phase, and it happens very quickly with the least timestamps.

2) Kernel Driver Initialization: Once the kernel core components are initialized, it begins to detect and initialize peripheral devices as described in the device tree. MMIO buses and GPIO interfaces are brought online, and power management components such as PMICs, regulators, and clocks are configured. Errors observed from the phase are attributed to missing components and are expected. For example, "`nau8822 3-001a: Failed to issue reset: -6`" occurs when the I2C driver fails to communicate with the NAU8822 audio codec, which we expect to see since there is no audio subsystem in the DUT. However, this is a substantially long phase and takes some time, and some drivers are misconfigured due to bit flips in memory but we can't observe the errors yet since the drivers are not being used.

3) Storage & Filesystem Initialization: After drivers with the corresponding peripherals, the kernel begins to configure the storage and filesystem, which involves initializing the flash memory, detecting partitions, and mounting the root filesystem. Journaling mechanisms also attempt to clean old journals retained from unclean shutdowns. We observe interesting errors at this stage, indicating storage failures that compromise system stability. For instance, "`blk_update_request: I/O error, dev mmcblk2, sector 17160 op 0x1:(WRITE)`" suggests that a write operation failed due to a previous driver issues, which could result in data corruption. Similarly, "`blk_update_request: I/O error, dev mmcblk2, sector 88248 op 0x0:(READ)`" indicates a read failure, possibly due to a failing flash memory chip or power instability, which may prevent loading essential files. Additionally, "`Buffer I/O error on dev mmcblk2p1, logical block 1121, lost async page write`" highlights that buffered writes failed, further increasing the risk of filesystem corruption and data loss. The errors from this stage cascade into corresponding file system errors when the file system is being mounted.

4) Filesystem Mounting/Remounting: In the final phase of kernel boot, the EXT4 filesystem is mounted, journal integrity is validated, and the system is handed over to user-space, where systemd or init is launched to load critical services. When filesystem corruption occurs, errors such as "`EXT4-fs error (device mmcblk2p1): _ext4_find_entry:1678`" indicate that essential metadata is unreadable, potentially preventing access to critical system files. The error "`Aborting journal on device mmcblk2p1-8.`" suggests that the EXT4 journaling system has encountered an irrecoverable failure due to the I/O write operations failing in the previous phase; this forces the system into entering a read-only state. Another related issue, "`JBD2: Error -5 detected when updating journal superblock for mmcblk2p1-8.`", occurs when the journaling subsystem fails to commit a transaction due to an I/O error, increasing the risk of filesystem corruption upon reboot. As a last resort, "`EXT4-fs (mmcblk2p1): Remounting filesystem read-only`" is triggered, forcing the file system to mount into read-only mode to prevent further corruption. While this may allow the system to continue booting, any write operations will fail, severely limiting functionality.

5) User-space Handoff: After the kernel "completes" its initialization, the init system in the user space takes over to start the system services essential for a working runtime environment. However, failures in the earlier kernel boot stages, result in corruption of the file system, i.e., the file system not being properly mounted or mounting in a read-only state, critical user-space services such as the greenboot

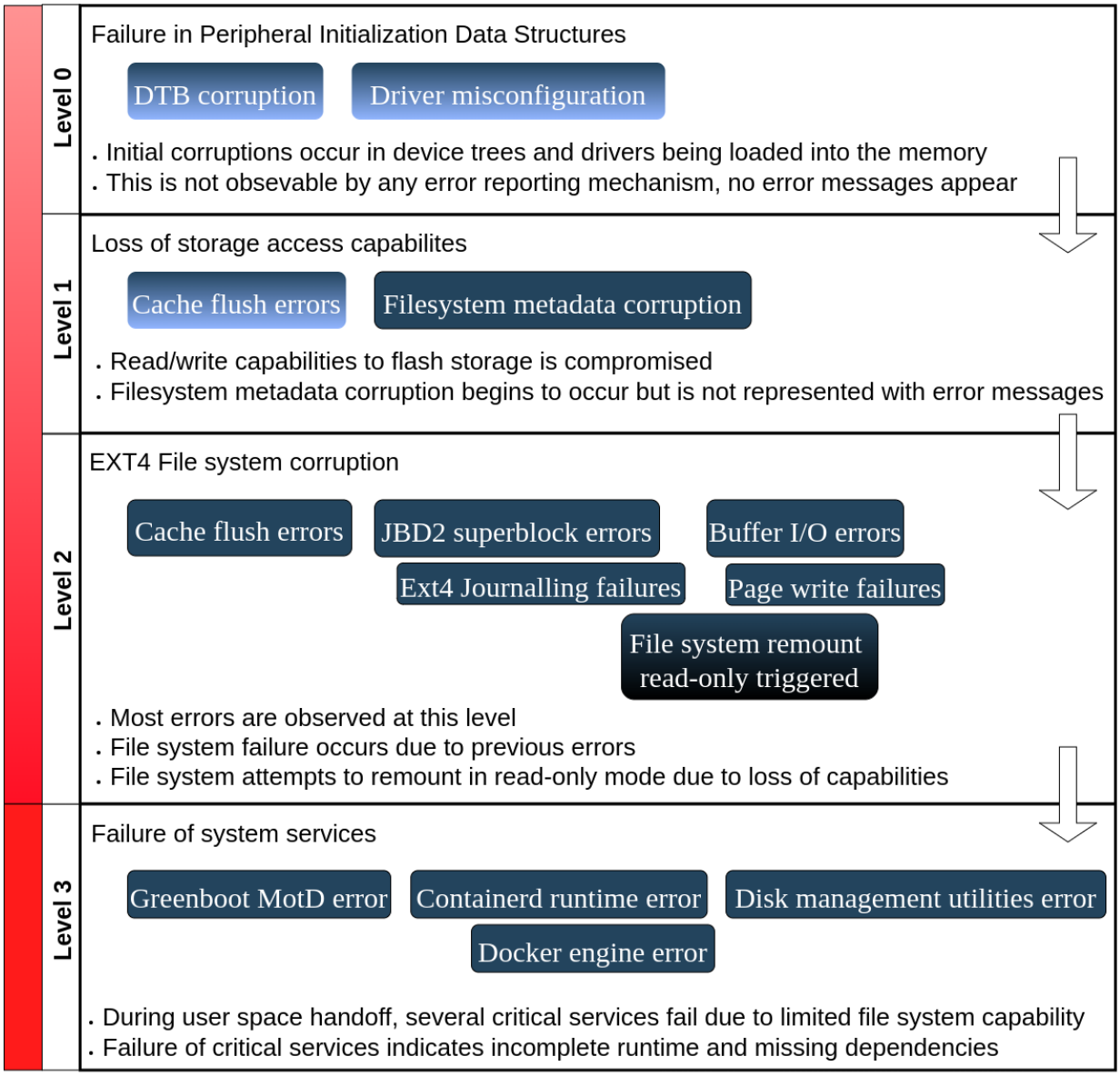

Fig. 5. Cascading mechanism of faults

MotD generator, containerd runtime, Docker engine, and disk management utilities fail to start. These failures occur because the dependency files and configurations required by these services are missing or inaccessible. In the end, the handover does not allow the user space to enter a working runtime environment.

V. Discussion

In the following, we discuss the errors observed and recommend actions for further improving the resilience of Linux during its boot phase.

A. External Boot Health Monitoring

Our power measurements in Section IV-A revealed interesting data, which can potentially be leveraged when externally observing the boot process of a compute module. Healthy boot sequences exhibit peaks within an expected power range, whereas failed boots tend to show prolonged instability or premature drop-offs. Successful boots should briefly reach peak power before stabilizing, whereas failures may result in continuous power fluctuations or abrupt shutdowns. Detecting these patterns in real-time could allow for early failure diagnostics and system intervention. Furthermore, power monitoring can provide insights into bootloader issues, kernel failures, and filesystem corruption based on characteristic power fluctuations. Implementing real-time power alerts over the expected power profiles could therefore serve as early warning signals, in particular during the phase where the kernel is not fully initialized.

B. Boot Logs

EXT4: Most faults that we observed in the boot logs point to failures in the ext4 file system and, in particular, to recovery actions triggered in response to failures. For example, the OS tries to remount the file systems, and if this fails again, it will try so in a read-only manner so as to not corrupt files. The root causes we could identify are bit flips that corrupt file data structures in memory. These corruptions often go unnoticed until a function that accesses them fails, either by finding a value that does not match the expected format, or by dereferencing pointers that have been corrupted and hence point to the wrong locations. RAM, thereby, turned out to be much more susceptible to the radiation levels we tested than flash, since the data on the flash memory remained intact and allowed subsequent boot attempts to succeed. We attribute the latter to the high error correction capacity that is inherent in the way flash memories are constructed today.

Normally, file system errors are relatively easy to fix, in particular corrupt blocks on disk, provided the log is intact and file structures, like the empty block list. In fact, tools like `fsck` (File System Consistency Check) perform these exact operations. In our observations, running the tool frequently caused the kernel to panic and was followed by a reboot, which indicates that the error is located in memory and only in a few cases on the flash drive.

We therefore recommend protecting the in-memory (i.e., RAM) file system data structures (e.g., by keeping redundant copies or by early on locating them in error corrected memory locations).

Bootloader failures A further source of failures originates from the bootloader not being able to locate the OS image or not being able to copy it from flash memory to RAM. These situations required a complete power cycling and reset of the device. This observation and the fact that the device finally ceased to operate indicate permanent memory failures in our accelerated test.

Since our tests primarily focused on Linux booting, further tests will be needed to identify how the bootloader itself can be made more robust to radiation faults. Aside from ECC, one option is to add further error checks in the bootloader code and to provide redundant information in the data structures and configuration files it uses.

C. Cascades

We observed that sometimes failures occur due to another failure that went unnoticed before. We call these effects to be *bleeding* into other subsystems. We observed bleeding when only a few static files became corrupted early during boot. We where then not able to observe the corruption directly, but only when a certain functionality invokes said files. As a result of such corrupted data structures, any functionalities, and their dependencies invoking the same resulted in failure, and as such, we observed many consequent failures in our logs. This is almost as if the previous errors bleed into the system to cause more failures, hence our choice of name.

To avoid these cascades, aside from a consequent use of ECC memory, which may be too costly, file, data structure, and functionality-specific mitigation strategies might be an interesting avenue for future research, in particular when they can be adapted dynamically to the radiation level perceived.

D. Summary

Overall, we observed the system to be significantly more vulnerable during the early boot phases, which suggests initializing and using the system's internal recovery mechanisms early on, in particular ECC. Once these mechanisms are up and running, reducing the frequency of costly actions (such as scrubbing) after the system is booted, should help reduce its boot time vulnerability.

Once the initial phase passed, most errors were concentrated in the file system and its in-memory data structures. More frequent checkpointing of those, error correction, and additional checks, possibly combined with a localized file system reset, should already be a significant advantage and help Linux overcome its boot phase.

VI. Conclusions and Future Work

In this paper, we report the findings of Linux booting on an iMX8M under up to 58 MeV proton radiation. We observed and evaluated the power output as well as the kernel logs while booting Linux several times at different energy levels. We further analyzed the root causes and discussed some countermeasures to increase the resilience of Linux's boot process under radiation. Among them are the early initialization and use of ECC memory if available to already benefit from correctable single bit faults, even if double bit faults cannot be handled. We recommend suppressing reboot on double-bit faults during boot and applying adaptive scrubbing techniques. We further discuss several opportunities at the level of the file system, which we found to be involved in most fault situations, as well as a few other recommendations. In particular, we found that once the system is properly initialized, it exhibits significantly more robustness during normal operation, which justifies the use of more costly mechanisms during boot and, at the same time, suggests mechanisms for later on that can be tuned to the application and the current radiation level that the system perceives.

References

[1] M. Bagatin and S. Gerardin, "Ionizing radiation effectsin electronics," 2016.

[2] D. M. Fleetwood, P. S. Winokur, and P. E. Dodd, "An overview of radiation effects on electronics in the space telecommunications environment," *Microelectronics Reliability*, vol. 40, no. 1, pp. 17–26, Jan. 2000.

[3] S. Duzellier, "Radiation effects on electronic devices in space," *Aerospace Science and Technology*, vol. 9, no. 1, pp. 93–99, Jan. 2005.

[4] M. V. O'Bryan, E. P. Wilcox, T. A. Carstens, *et al.*, "NASA Goddard Space Flight Center's Current Radiation Effects Test Results," en, in *2023 IEEE Radiation Effects Data Workshop (REDW) (in conjunction with 2023 NSREC)*, Kansas City, MO, USA: IEEE, Jul. 2023, pp. 1–13.

[5] B. Todd and S. Uznanski, *Radiation Risks and Mitigation in Electronic Systems*, arXiv:1607.01573 [physics], 2015.

[6] M. S. Reorda, "A Low-Cost SEE Mitigation Solution for Soft-Processors Embedded in Systems on Programmable Chips," en,

[7] A. Pouponnot, "Strategic use of SEE mitigation techniques for the development of the ESA microprocessors: Past, present, and future," in *11th IEEE International On-Line Testing Symposium*, ISSN: 1942-9401, Jul. 2005, pp. 319–323.

[8] S. Guertin, S. Vartanian, and A. Daniel, *Raspberry Pi Zero and 3B+ SEE and TID Test Results*, en.

[9] S. Memon, R. Graczyk, T. Rajkowski, J. Swakon, D. Wrobel, S. Kusyk, and M. Papadakis, *When Radiation Meets Linux: Analyzing Soft Errors in Linux on COTS SoCs under Proton Irradiation*, arXiv:2503.03722 [cs], Mar. 2025.

[10] S. Guertin, *A53 SEE Performance in Raspberry Pi and MX8M*, en.

[11] L. Frias-Dominguez, G. Leon, J. M. Badia, J. A. Belloch, M. Garcia-Valderas, A. Lindoso, and L. Entrena, "Radiation reliability of system reboots in commercial off-the-shelf SoC," *IEEE Transactions on Nuclear Science*, pp. 1–1, 2025, Conference Name: IEEE Transactions on Nuclear Science.

[12] E. Miller, C. Heistand, and D. Mishra, "Space-operating linux: An operating system for computer vision on commercial-grade equipment in leo," in *2023 IEEE Aerospace Conference*, 2023, pp. 1–12.

[13] J. R. Sklaroff, "Redundancy management technique for space shuttle computers," *IBM Journal of Research and Development*, vol. 20, no. 1, pp. 20–28, 1976.

[14] D. R. Turner, "Radiation shielding.," en,

[15] N. Dodds, N. Hooten, R. Reed, *et al.*, "Effectiveness of sel hardening strategies and the latchup domino effect," *IEEE Transactions on Nuclear Science*, vol. 59, pp. 2642–2650, Dec. 2012.

[16] R. W. Hamming, "Error detecting and error correcting codes," *The Bell system technical journal*, vol. 29, no. 2, pp. 147–160, 1950.

[17] C.-L. Chen and M. Hsiao, "Error-correcting codes for semiconductor memory applications: A state-of-the-art review," *IBM Journal of Research and development*, vol. 28, no. 2, pp. 124–134, 1984.

[18] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo, "Resilient Intrusion Tolerance through Proactive and Reactive Recovery," in *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, Dec. 2007, pp. 373–380.

[19] A. Matovic, R. Graczyk, F. Lucchetti, and M. Völp, "Consensual resilient control: Stateless recovery of stateful controllers," in *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2023, pp. 14–1.

[20] P. C. Anderson, F. J. Rich, and S. Borisov, "Mapping the South Atlantic Anomaly continuously over 27 years," *Journal of Atmospheric and Solar-Terrestrial Physics*, Dynamics of the Sun-Earth System: Recent Observations and Predictions, vol. 177, pp. 237–246, Oct. 2018.

[21] G. D. Badhwar, "The Radiation Environment in Low-Earth Orbit," *Radiation Research*, vol. 148, no. 5s, S3–S10, Nov. 1997.