Porting NASA's core Flight System to the Formally Verified seL4 Microkernel

Juliana Furgala, Samuel Jero, Andrea Lin, Richard Skowyra
MIT Lincoln Laboratory
{juliana.furgala, samuel.jero, andrea.lin, richard.skowyra}@ll.mit.edu

Abstract—**Satellite systems provide crucial services for the modern world, including global position, navigation, and timing as well as world-wide communication, earth imaging for weather forecasting, and a host of other functions. Due to the critical nature of these services and their increasing importance, satellites are increasingly targeted by attackers, including both criminals and nation-state actors. Unfortunately, the software controlling these satellites has not been designed with security in mind due to the assumption that access is difficult. With the increasing commodification of space, that assumption no longer holds, leaving these systems exposed and vulnerable.**

In this paper, we share our experience attempting to combine real flight software with a key security technology developed by the security community. In particular, our goal is to run NASA's core Flight System (cFS) on top of the formally verified seL4 microkernel to eliminate vulnerabilities related to the operating system and provide a strong foundation for satellite software systems. While we were successful at doing so, it required more than a year of effort and the development of a significant set of operating system services beyond the seL4 microkernel. Along the way, we learned some key lessons about flight software and security technologies like seL4.

I. Introduction

Today, satellites provide essential capabilities for modern life, including global positioning and navigation provided by Global Navigation Satellite System (GNSS) constellations, meteorological imaging for weather prediction, and world-wide communication with hand-held equipment. GNSS constellations are essential for everything from navigating a new city using your smart phone to ensuring accurate time on everything from trains to ATMs. Proliferated Low Earth Orbit (pLEO) constellations, like Starlink, provide low-latency, high-throughput Internet access almost anywhere in the world. More recently, we have seen the power of earth imagery and RF data collected by satellites in near-real time to understand the conflicts in both Ukraine [1], [2] and Iran [3].

Since satellites provide such critical services, it should come as no surprise that we are increasingly seeing attacks targeting these services and the satellites providing them. Historically, satellites were extremely custom systems, and it required complex, expensive equipment to communicate with them. However, the increasing commodification of space and increasing reliance on Commercial-Off-The-Shelf (COTS) components make satellites much more accessible to attackers. At this point, it is routine to experience GNSS jamming in and around contested areas [4], and nation-state actors have demonstrated willingness to target satellite systems, as evidenced by Russia's attack on the Viasat KA-SAT system immediately prior to their invasion of Ukraine [5]. Since that time, we have seen further attacks on Viasat by Salt Typhoon [6] and government targeting of Starlink [7]. Competitions like DARPA's Hack-A-Sat have also provided insight into what a motivated attacker could do against a satellite [8]. All of this has left operators of satellite systems increasingly concerned about the cybersecurity of their systems.

Satellites operate in a physically hostile environment with dramatic temperature swings and significant cosmic radiation and are usually significantly constrained by Size, Weight, and Power (SWaP). They also operate under the unique constraint that once launched there is effectively no possibility of human intervention during their entire operational lifespan, which is usually years if not decades. Prior work has explored these challenges and their implications in significant detail [9], [10].

As a result of these challenges, flight software for these systems tends be written in low level languages like C/C++ and to rely on specialized proprietary real-time operating systems, often with custom platform support. Although this software is carefully designed and tested to ensure correct behavior under benign failures, little effort is put into ensuring correct behavior in the face of malicious adversaries. In the cases where this software was available to researchers for examination, both fuzzing [11], [12] and manual analysis [13] have revealed a significant number of critical vulnerabilities.

Security practitioners and researchers have developed many tools, techniques, and systems that could potentially be de-

Workshop on Security of Space and Satellite Systems (SpaceSec) 2026
23 February 2026, San Diego, CA, USA
ISBN 978-1-970672-02-2
https://dx.doi.org/10.14722/spacesec.2026.23003
www.ndss-symposium.org

ployed on satellites to eliminate or mitigate the current vulnerabilities and create more secure flight software for satellites. These techniques include modern memory safe languages like Rust, techniques like CFI [14] and ShadowStacks [15] to mitigate vulnerabilities in unsafe languages, isolation approaches like Software Fault Isolation [16] and HAKCs [17], and formally verified parsers [18] and microkernels [19]. Not all of these techniques will be appropriate or mature enough for highly embedded systems like satellites, but some of them are likely to be both suitable and beneficial.

In this paper, we share our experience attempting to combine real satellite flight software with an operating system kernel developed and formally verified by the security community. In particular, we focus on running NASA's core Flight System (cFS) [20], a leading flight software that has been used in dozens of missions over 20 years, on the formally verified seL4 microkernel [19]. Successfully doing this would dramatically improve the security of satellite systems by removing or reducing the impact of vulnerabilities in the operating system.

While we were able to successfully demonstrate cFS running on seL4, it required more than a year of effort and the development of a significant set of operating system services beyond just the seL4 microkernel. While these services are not verified and thus likely contain bugs, the separation of the operating system into multiple components itself provides significant security improvements [21], [22]. We write these services in Rust to minimize the risk of memory safety vulnerabilities.

We also document a number of key lessons learned about satellite flight software and systems security technologies, including the surprising dynamism of satellite flight software, the impacts of availability as a key priority on software design, the importance of designing flight software with multiple components, and the complexity of microkernel operating systems.

The remainder of this paper is organized as follows. Section II provides background on flight software, NASA's core Flight System, and seL4. Section III then presents our analysis of cFS to understand its requirements and assumptions. We then describe our effort to port cFS to seL4 and the design of our operating system services in section IV. Section V presents an evaluation of the performance of our system while section VI describes the lessons learned from this effort and section VII offers some discussion other OSes used on space vehicles. Finally, we conclude in section VIII.

II. Background

This section provides background on satellite flight software in general, NASA's core Flight System in particular, and the seL4 microkernel.

A. Satellite Flight Software

Satellite flight software is effectively the autonomic nervous system of a satellite. It is responsible for keeping the space vehicle operational as well as maintaining communication with the ground, including sending telemetry and processing commands. Keeping the space vehicle operational involves ensuring sufficient power supply; the flight software makes orientation decisions to ensure that batteries charge sufficiently, and it controls what components of the vehicle are powered on and drawing power at any given moment. Another major component of ensuring that the space vehicle remains operational is controlling the orientation of the vehicle using sensors and actuators like reaction wheels. This also involves significant complexity as changes in orientation may be necessary to achieve specific mission goals or to avoid physical damage, for instance, by pointing a camera at the sun.

The flight software also provides the primary means for operators on the ground to interact with the satellite, by sending telemetry about the performance of the spacecraft and receiving control commands from the ground and acting on them. It often also controls the transmission of mission data, like imagery or sensor data.

Because of the critical role that flight software plays in satellites, it also often includes one or more safe modes. These are modes of operation that are entered when unexpected or erroneous conditions occur, often as a result of a hardware failure or programming bugs. These modes are designed to ensure that the satellite remains safe, conserves battery power, and is able to communicate with the ground, so that operators can examine the situation and work to correct it.

Unfortunately, there is a growing body of work indicating that satellite flight software is also extremely vulnerable to adversarial attack. While historically expensive radio equipment might have been required to send data to these satellites and opportunities for supply chain attacks were rare, this is no longer the case. Consumer software defined radios have dramatically lowered the barrier to entry. Further, it is increasingly common for satellites to reuse significant quantities of existing software, both open-source and proprietary. Prior work examining open source flight software has confirmed this general state of vulnerability [11]–[13].

B. NASA's core Flight System (cFS)

NASA cFS is a widely used and open-source flight software framework developed by NASA Goddard. It has been used in over 40 missions by multiple different space agencies and across a variety of different space systems including satellites, landers, rovers, crew habitats, and even space suits [23]. Originally developed in 2004 as a modular, reusable system to enable NASA to develop software for two major missions simultaneously [24], it since grown into framework offering a core set of standard components and many optional applications to help missions quickly build out their needed software stack [25].

cFS is written in C and based on a message bus architecture. A small set of services make up the core Flight Executive, or the core of the system. These include the software data bus, an executive service responsible for launching and managing other applications, time services, event services, and table services, which are responsible for storing adjustable parameters for other applications in tables conceptually similar

TABLE I
CAPABILITIES IN seL4

Capability Type	Description
Untyped	Raw memory to create other capabilities
Page	A page (technically frame) of memory
Page Table	A page table
TCB	A thread control block for a thread
CNode	A table that holds capabilities
Endpoint	A synchronous communication endpoint used for IPC
Notification	An asynchronous communication primitive somewhat similar to an array of mutexes
Interrupt	Controls access to and configuration of an interrupt
I/O port	Controls access to I/O ports (x86 only)
ASID	Controls access to Address Space IDs in the TLB

to a database [25]. Around this core a variety of optional apps are provided to support periodic housekeeping telemetry, scheduled commands, communication protocols like CCSDS File Delivery Protocol (CFDP), and many other features. Finally, missions usually add custom apps to cFS to manage their particular mission payloads and functionality.

C. seL4

seL4 is a formally verified microkernel of L4 heritage [19], [26]. Its proofs guarantee functional correctness. This correctness property means that seL4's code correctly implements the specification describing its behavior. Importantly, because the specification does not describe things like buffer overflows and panics, this proves the code free from large classes of vulnerabilities and bugs. Additionally, seL4's proofs guarantee binary correctness and ensure that its API provides integrity. seL4 is one of the largest formal verification efforts to date, requiring over 30 person years of effort to verify its roughly 9,000 lines of C code [19].

seL4's L4 heritage means that it offers extremely fast IPC, unlike older microkernel designs that developed a reputation for poor performance [27]. It also uses capabilities as its means of controlling access to system resources. Capabilities are, formally, unforgeable tokens of authority [28]. In other words, they are handles used to control resources, in a manner very similar to a UNIX file descriptor. Table I lists the capabilities that exist in seL4.

It's also important to realize that seL4 is a microkernel. It only provides functionality that must be provided in kernel mode, leaving many key aspects of operating systems (OSes) to userspace. In particular, seL4 provides scheduling and address spaces, leaving all other parts of operating system functionality, like executable loading, device drivers, and networking to userspace. To help with all of this functionality, seL4-based systems often use CAmkES [29] or, more recently, Microkit [30] to load a set of processes and connect them together into a functioning system. These systems take a configuration describing a set of executable processes that correspond via IPC or shared memory. At boot, they then handle the process of creating and loading the defined processes and connecting them together as specified.

III. ANALYSIS OF cFS

In this section, we explore the architecture of cFS in more detail and describe our initial analysis of cFS and the operating system functionality it requires.

As previously discussed, cFS is designed with a message bus architecture. In other words, all cFS applications connect to a message bus and can send or receive messages from other applications using this bus. In this way, applications can communicate with each other or with the ground as needed. Figure 1 illustrates this design. It shows the core cFS services, called the core Flight Executive (cFE), along with other standard cFS apps and mission apps all connected to the message bus. These core services include the message bus itself (SB), an executive responsible for launching and controlling all other applications, time services responsible for keeping track of time, event services responsible for key system events, and table services, which is similar to a simple database and responsible for storing adjustable parameters for other applications [25]. Other illustrated apps include the housekeeping app for providing regular status messages to be sent to the ground, a scheduled commands app for running commands at particular moments in time, and a limit checker app for ensuring that parameters stay within defined limits and issuing alarms if they do not. Mission-specific apps, like Command Ingest and Telemetry Output, handle interfacing with mission hardware, like sending telementry or receiving commands over the radio.

From a software architecture perspective, cFS is built with a number of layers and abstractions, as illustrated in Figure 2. cFS assumes that some operating system is running on the space vehicle's flight computer. In fact, cFS supports a wide variety of operating systems, including Linux, VxWorks, and RTEMS. To accomplish this, cFS introduces two abstraction layers: the Operating System Abstraction Layer (OSAL) and the Platform Support Package (PSP). These layers describe a system-agnostic API for operating system functionality required by cFS and then implement that API for each of the supported operating systems.

cFS is implemented in C as a single-process architecture with many threads. Each application in cFS consists of one or more threads. This design makes the message bus architecture extremely efficient, as only pointers to data need to be passed around. However, it also makes the system extremely complicated as all applications live in the same address space. cFS also makes several unusual design decisions to ensure that it can operate in real-time environments. For instance, cFS does not allocate heap memory at all, as asking the OS for memory could take a significant amount time in the worst case, or might even fail. Instead, cFS prefers to use global variables and fixed size arrays of objects. Although unusual, this approach is frequently used in other real-time systems, including other flight software.

As the first step of porting cFS to run on seL4, we set out to characterize what functionality a minimal cFS setup requires from the operating system. We used cFS version 7.0.0-rc4

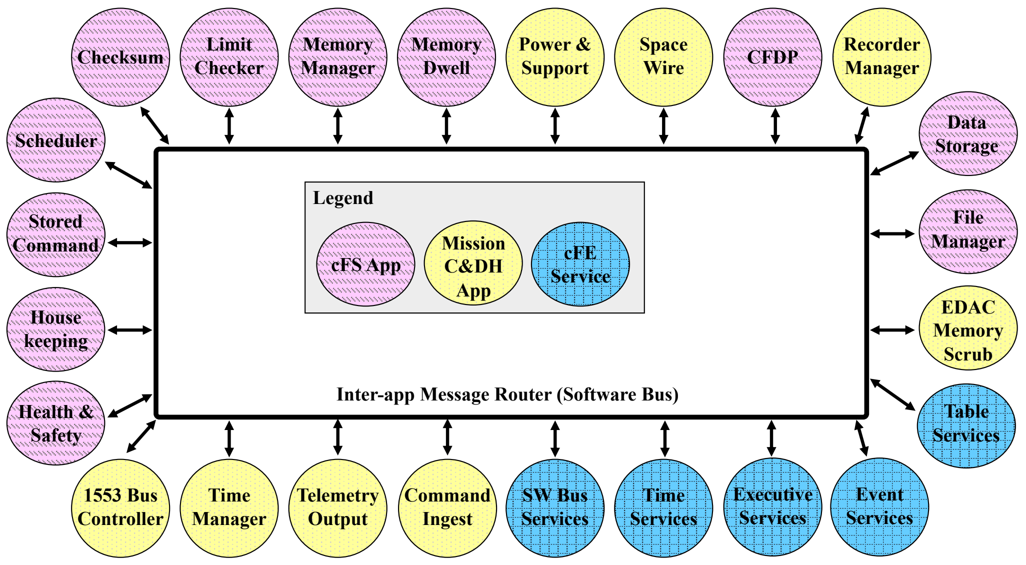

Fig. 1. A typical cFS system, from [25]

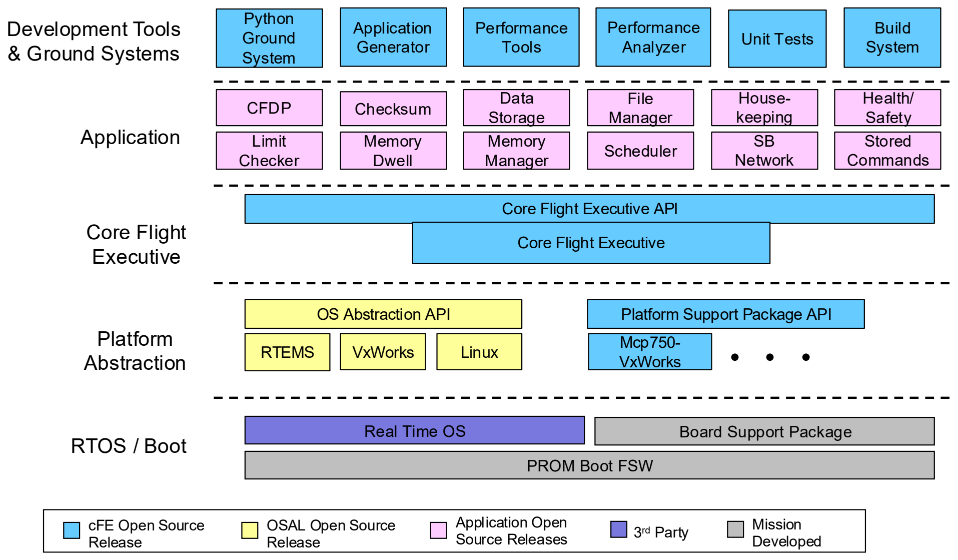

Fig. 2. Software Architecture of cFS, from [25]

and focused on a minimal system containing the five core cFS services plus the housekeeping app, which sends periodic status messages, and test versions of the Command Ingest and Telemetry Output apps. This minimal cFS setup will be the focus of the remainder of this paper.

We characterized the required functionality by examining the OSAL to understand what OS functionality it contained and by examining which parts of that were called by our minimal set of apps. Our analysis revealed approximately 36 types of operating system functionality required by this minimal cFS setup. This functionality covered a range of common OS primitives, including threading, message queues, timers, and synchronization primitives (*e.g.,* mutexes and semaphores). The details are shown in Table II. Note that although it is possible to use filesystems and networking in cFS apps, our paired down setup does not require these calls, which dramatically simplified our efforts.

IV. Porting cFS to seL4

This section describes our experience porting cFS to seL4. We targeted the armv7a Xilinx Zed Board (ZC702) development kit as our target hardware because it is both well supported by seL4 and has previously flown in space. As mentioned before, we use cFS version 7.0.0-rc4.

TABLE II
OS FUNCTIONALITY REQUIRED BY CFS

Operation Type	Operations
threading	create, destroy, exit, delay, set priority, get id, my id, match
binary semaphores	create, give, take, timed wait, delete, get info
counting semaphores	create, give, take, timed wait, delete, get info
mutex	create, give, take, delete, get info
message queues	create, delete, receive, send
time	get current time
timers	create, start, cancel, delete, lock, unlock
I/O	print

We started this effort using all of the existing tooling surrounding seL4 including CAmkES [29][1] and the unverified support libraries written in C. CAmkES is a component based framework that statically defines a set of processes/threads in a system and the connections between them. It assumes the use of tens of thousands of lines of unverified C libraries to help C applications do things like allocate memory or manage seL4 capabilities.

In terms of the operating system functionality that cFS requires, it is actually fairly possible to map these to seL4 capabilities. A thread corresponds to seL4's TCB (Thread Control Block) capability. Semaphores and mutexes can be created using a couple of variables and an seL4 notification object capability. A message queue is a chunk of memory and a semaphore. Time and Timers are a little harder because they require the implementation of a device driver to access and configure hardware timers. However, CAmkES actually includes this functionality for several common platforms. Similarly, I/O requires a UART driver, which is also available in CAmkES.

The major issue we ran into was that CAmkES requires all of its resources to be specified statically up front in a configuration file. This fits poorly with the design of cFS, where all of these operating system resources are expected to be created and destroyed dynamically. cFS does have a maximum number of these resources that can be in use at any given time, but because resources can be created and destroyed, the total number is actually unbounded. Our solution to this was to use dynamic analysis to determine how many of each resource (e.g., threads, semaphores, mutexes, *etc.*) were actually used. We could then adjust the CAmkES configuration file to include the appropriate number of resources, allowing cFS to run on seL4.

While this approach achieved our goal of running cFS on seL4, it was unsatisfactory for several reasons. First, this mismatch between cFS's dynamic approach to resources and CAmkES very static approach is concerning. Although we could determine an appropriate number of resources using dynamic analysis, there is no guarantee that some rare code path would not require more resources, which would lead to a system crash. We would also have to perform this dynamic analysis every time we added additional apps or functionality to the system, leading to increasing likelihood of missing interactions and code paths as the system gets more complex. For flight software that may need to operate unattended for years, and for a domain where physical access to the system post-launch is impossible, these seem like unacceptable risks.

[1]We noted earlier that CAmkES has since been replaced with Microkit [30]; however, this initial effort predated the development of Microkit by several years. Further, Microkit follows the same static design principles, which ultimately caused us problems as discussed later on.

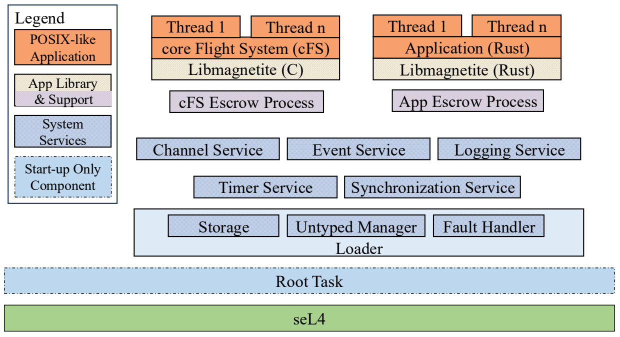

Fig. 3. Architecture of our Magnetite operating system

Second, we were concerned with the quantity of unverified C code present in CAmkES-based seL4 systems. Ultimately, we had several tens of thousands of lines of unverified C in our system. Since the seL4 kernel is only about 9,000 lines of C, we actually end up with more unverified C code than verified C code in our operating system. Of course, this unverified code is less critical than and is isolated by the verified code. However, the entire point of porting cFS to seL4 was to improve the security of the operating system by leveraging formal verification.

These issues lead us to return to the drawing board and consider what a more appropriate architecture might look like. As a result, we decided that actually building a new operating system around seL4 was necessary. By building a new operating system, we could support the dynamic creation and deletion of resources that cFS expects, thereby mitigating our first major problem. Further, our new operating system, which we call Magnetite, could be written in Rust. While we are not able to formally verify the entire operating system, at least at this moment, Rust enables us to mitigate the memory safety vulnerabilities that have historically plagued operating systems and similar systems software.

The architecture of our Magnetite operating system is shown in Figure 3. It is largely designed around providing the operating system functionality required by cFS while being able to support multiple applications and following the principle of least privilege [21], [22]. We continue to use seL4, but remove CAmkES entirely. Instead, we built an entirely new, short-lived root task and set of system services. Motivated by the principle of least privilege, we separate out the required operating system functionality into multiple services, with each type of functionality handled by its own service. In this way, the compromise of one service would not allow attackers

TABLE III
COMPARISON OF MAGNETITE TO REAL-TIME LINUX ON MICROBENCHMARKS. NUMBERS IN CPU CYCLES.

	Magnetite				Linux			
	Avg	Std Dev	95%tile	Max	Avg	Std Dev	95%tile	Max
Context Switch: Thread	504	0	504	550	1,061	25	1,077	3,232
Context Switch: Process	498	1	498	599	4,817	327	4,858	17,919
Round Trip IPC	1,136	3	1,137	1,241	-	-	-	-
Event Latency: equal prio	8,788	185	9,095	10,393	-	-	-	-
Event Latency: L2H prio	8,790	181	9,093	9,870	-	-	-	-
Event Latency: H2L prio	14,138	292	14,613	17,614	-	-	-	-
Mutex Uncontended	6,301	292	6,745	8,615	217	2	217	328
Mutex Contended	15,574	285	16,042	17,394	15,844	619	16,263	30,570
Semaphore Uncontended	5,360	200	5,689	6,348	117	90	116	9,112
Semaphore Contended	11,661	250	12,070	12,741	6,714	404	6,994	22,136
Timer Latency	12,202	210	12,536	13,907	20,666	1,068	21,171	33,118
Timer Latency w/ timerfd					6,494	632	6,842	14,806
Channel Latency: L2H prio	18,367	286	18,850	20,038	9,439	423	9,627	22,671
Channel Latency: H2L prio	18,505	273	18,983	20,271	11,508	841	11,711	71,169

to trivially pivot to other parts of the operating system.

The bottom layer of our operating system is its root task. This is the component that seL4 starts once it finishes booting. Its job is to standardize the system and hand off to the loader. The loader is the largest and most important of Magnetite's six core services. It is responsible for managing processes, threads, and memory allocation as for handling the loading of all other services and applications in the system. To do this, the loader exposes an API for creating and destroying threads that is used by cFS's OSAL and another for memory allocation used by other services. The loader also starts two threads to handle seL4 capabilities and faults that occur in system services.

The other five services each provide one particular type of functionality for applications. The synchronization service, for instance, provides semaphore and mutex primitives for use by applications. We implement mutexes with priority inheritance to support cFS. This means that if a low priority thread locks a mutex and then a higher priority thread blocks waiting for the mutex, the low priority thread temporarily has its priority raised until it is done with the mutex. This is a crucial feature for real-time systems, to prevent a low priority computation from stalling a high priority computation. It is also a feature that we did not support in our first implementation with CAmkES.

The timer and logging services provide device drivers for hardware timers and serial devices, respectively. They then expose those devices to applications over special APIs that cFS can call. The timer service is slightly more complex, because it has to multiplex a potentially arbitrary number of user timers onto a limited set of hardware timers. We use a timer wheel construction to do this, where user timers are binned together and triggered using a single periodic hardware timer.

Similarly, the channel service provides message queues to applications. These queues operate by sending messages to the service, which then queues those messages for the receiving application. These queues have configurable message size and queue depth. Finally, the event service allows applications to wait on any one of multiple events to occur, like a message being available in a queue or a timer firing. Other services notify the event service of these events, which can then trigger waiting applications.

All of these services provide APIs over InterProcess Communication (IPC). To enable applications to easily call these APIs, Magnetite provides libraries in C and Rust that wrap these IPC calls in simple function calls. Because cFS is designed to support multiple operating systems, it was a simple matter to add a new OSAL backend for Magnetite that makes the appropriate API calls.

With all of these services and libraries implemented, we were then able to run cFS on seL4 using our Magnetite operating system. We found that we were able to add additional apps to cFS without modifying Magnetite, validating our choice to create an operating system to support dynamic resources. We have subsequently found that other flight software like CHSS [31] and F-prime [32] make similar assumptions about being able to create resources dynamically. This validates the need for this functionality and our creation of a new operating system supporting it.

Creating an operating system, even a very simple one, is no small task. Our initial version of Magnetite, capable of running cFS, took a team of three developers almost a year and a half to create. Since then, we have continued to expand Magnetite with more functionality. It now provides shared memory, networking, and a filesystem as well as a significantly expanded set of device drivers. Additionally, it supports the aarch64 and x86_64 architectures in addition to armv7a.

Overall, our experience demonstrates that it is possible to adapt academic security technologies like seL4 for use in satellite flight software. However, it also highlights that doing so is not an easy task.

V. EVALUATION

In this section we evaluate the performance of Magnetite, our new operating system on seL4. Our goal is to understand the possible impact of introducing a secure operating system under cFS.

We use a series of microbenchmarks to compare the performance of key operating system operations utilized by cFS, including context switches, mutex locking/unlocking, semaphore operations, and timer latencies. As our baseline, we use the Linux kernel with the `PREMPT_RT` patch, a configuration usually called real-time Linux. Although dedicated real-time operating systems (RTOSes) are more common for space systems, they are usually either extremely expensive or difficult to configure. However, Linux has been used in space systems [33], and its real-time performance is acceptable in many other domains.

We use a Zynq-7000 XC7Z020 SoC for our evaluation. It includes a dual-core Arm Cortex-A9 processor running at 667 MHz and a Xilinx FPGA. We use only a single core for this evaluation and do not use the FPGA at all. Our benchmarks are written in C and compiled with gcc version 9.4.0; Magnetite uses Rust version nightly-2022-07-14. We compare against Linux kernel version is 5.4.61-rt37. Each result we report is computed from 10,000 test runs.

Table III shows the results of our evaluation. We see that Magnetite has extremely fast context switches that are almost twice as fast as Linux on average. For mutexes and semaphores, Magnetite has similar average-case overhead when contended, but is slower for uncontended operations. This is a side-effect of our implementation of priority-inheritance. In particular, while Linux is able to implement mutex operations in a few instructions when uncontended, priority-inheritance requires Magnetite to do an IPC operation to our synchronization service.

For timers, our results reveal an interesting situation. Magnetite has much better latency than the standard POSIX interface for timers. However, Linux has a special interface called `timerfd` that is faster still. cFS chose to use the POSIX standard interface, presumably because it is standard. This means that cFS on Magnetite sees better timer performance than on Linux. For channels, or message queues, we see that Linux is faster in the average case, but Magnetite actually has a better worst case performance.

Overall, we see surprisingly similar performance between Magnetite and Linux for operations that cFS cares about. Average case behavior is rather mixed; however, Magnetite usually performs better in the worst case. This suggests that cFS on Magnetite would be practical and may even improve worst case performance.

VI. Lessons Learned

Over the course of this effort, we identified several key lessons learned about satellite flight software, its security, and microkernel-based systems. This section describes and discusses these lessons in detail.

Flight software is surprisingly dynamic. Our initial assumption going into this effort was that satellite flight software was very static. We assumed that controlling a satellite would involve a fixed set of tasks that do not change, so there would be no need for significant changes in software functionality. Hence, the flight software could probably allocate a fixed set of resources at startup.

This turns out to be completely wrong. Applications in cFS can be started and stopped or added and removed from the system at runtime. Similarly, operating system resources like message queues, semaphores, timers, and sockets can be dynamically created or deleted as cFS applications change their behaviors. This supports missions that are surprisingly dynamic, with different tasks needing to be done at different times and often with updates and fixes occurring to parts of the system but not to others during the course of the mission. For example, a landing rover or actively launching satellite will have different needs than a satellite currently collecting data in the course of a mission or hosting multiple missions.

This dynamic approach conflicts with the usual assumptions about real-time systems in the security community. In particular, frameworks like CAmkES and Microkit assume that a system has a fixed set of tasks that require fixed resources and communicate in fixed patterns. Perhaps that is true for a engine control unit in a car responsible for controlling braking behavior, but it is completely incorrect for flight software. Our Magnetite operating system builds a dynamic layer on top of seL4 that enables these kinds of high reliability dynamic systems.

Availability is an overarching priority. Flight software turns the usual Confidentiality, Integrity, Availability triad on its head, prioritizing availability above all. While this is also common in cyber-physical systems [34], flight software has unique challenges here because loss of availability usually means termination of the mission since physical access is effectively impossible.

What surprised us was how this focus on priority impacted even the APIs exposed by cFS in its OSAL. For instance, the majority of operations within cFS have timeouts to prevent deadlock and allow careful scheduling of hard real-time tasks. This also extends to the use of priority-inheritance for mutexes to avoid priority inversion, where a low priority task delays a high priority task, as discussed in section IV.

Flight software often lacks isolation between components. Although we focus in this work on securing the operating system below cFS, our analysis of cFS revealed a troubling design from a security perspective. In particular, all of cFS—the core services and applications—is a single address space. Each application or service is one or more threads running within that shared address space.

While this allows for performant exchange of information between apps, it provides no isolation between different cFS apps or services. As such, a vulnerability in any app trivially compromises the entirety of cFS. Even if other apps do not have vulnerabilities, the attacker can trivially manipulate data and code for all apps in the system. This is not a hypothetical concern either; recent research shows that vulnerabilities do, in fact, exist in cFS [13].

At the operating system level, Magnetite illustrates one approach to mitigate this issue. Magnetite is designed as a set of services, each of which is a separate process. These services communicate via IPC or shared memory, but maintain their isolated nature. As a result, a compromise in one service does not compromise the entire operating system. A similar approach to flight software might keep the central data bus found in cFS, but move each core service or app into a separate process. This appears to be the goal of the CHSS flight software [31].

Microkernel operating systems inherit all the problems of distributed systems. Our experience building Magnetite sheds significant light on why all major operating systems are monolithic designs, despite the significant security benefit that a microkernel design provides. While separating the operating system into many interacting services reduces the impact of compromise, it also introduces the numerous problems of distributed systems to operating systems.

Once the number of interacting services grows beyond two or three, it is surprisingly easy to establish paths where service A calls service B which calls service C which calls service A, resulting in a deadlock. Careful design and planning is needed to avoid these kind of cycles. We ran into this issue more than once while building Magnetite. Even just ensuring that time-bound operations traverse only a small number of services can be challenging.

A closely related issue is the way that different parts of an operating system interact. At first glance, a filesystem has no reason to interact with timers and time, yet filesystems include modification and update timestamps and often rely on time to flush filesystem caches.

Similarly, sending and receiving messages is key to systems made up of multiple isolated components. Unfortunately, we know that message parsing and formatting code is particularly bug prone. This means that there is now much more of this bug-prone code in the entire system compared to a monolithic operating system. Techniques that can automate the serializing and deserializing of messages can help here, but some of this complexity is seemingly unavoidable.

VII. Discussion

This section discusses other Real Time Operating Systems (RTOSes) that are often used in space vehicles and their security challenges. It also considers future improvements to Magnetite.

A. Other RTOSes

Although the earliest space systems used custom operating systems, most space vehicles today use existing commercial RTOSes, like RTEMS [35] or VxWorks [36]. Unfortunately, while these systems are carefully designed to provide the predictable performance demanded by real-time systems, they are significantly lacking from a security standpoint. RTEMS in particular is a single address space operating system that provides no memory permissions nor any separation between user and kernel code [37]. As a result, RTEMS provides no isolation either between user-space components or between the user-space and the kernel. Hence, any compromise in any part of the system compromises everything.

While other operating systems used in space vehicles, like VxWorks or real-time Linux, an increasingly popular option, do provide isolation between components, they are monolithic in design. In other words, the entire operating system is a single component and any compromise in the operating system enables an attacker to manipulate the entire operating system.

In contrast, Magnetite is a microkernel-based system. seL4's formal proofs of correctness mean that the most privileged, most important part of the system is both from free large classes of bugs and functions correctly. For the rest of the components of Magnetite, the separation into multiple, isolated components means that a vulnerability in one of those components only impacts that component. To pivot between components, an adversary would need to find a vulnerability in each desired component.

B. Future Magnetite Improvements

While Magnetite is written in Rust and separated into many isolated processes, further security improvements are possible. Of particular note, the Magnetite services have not been verified, which leaves open the possibility of vulnerabilities or correctness issues. Future work could consider the practicality of verifying these services.

VIII. Conclusion

Satellite systems are essential to modern life, yet face many challenges due to their harsh operating environment, lack of physical access after launch, and increasing attention by potential attackers. Unfortunately, the flight software controlling these space vehicles has not been designed with security in mind, leaving these systems exposed and vulnerable. We attempted to run NASA's cFS flight software on the formally verified seL4 microkernel, with the goal of eliminating vulnerabilities related to the operating system. We found that the existing tooling for high assurance systems, like CAmkES, was not suitable for use with cFS. Instead, we developed our own set of operating system services around seL4, requiring more than a year of effort. We hope that this effort will inspire further work on combining existing security techniques with flight software.

References

[1] D. Werner, "Hawkeye 360 detects gps interference in ukraine," Jan 2023. [Online]. Available: https://spacenews.com/hawkeye-360-gps-ukr/

[2] M. Torrieri, "How satellite imagery magnified ukraine to the world," Jul 2023. [Online]. Available: https://interactive.satellitetoday.com/via/articles/how-satellite-imagery-magnified-ukraine-to-the-world

[3] The Associated Press, "Satellite photos reveal damage to iranian missile bases and nuclear facilities after israeli strikes," Jun 2025. [Online]. Available: https://apnews.com/photo-gallery/iran-israel-missile-bases-satellite-photos-c2ca6d9d80567403cdb46653ffb17eb0

[4] M. Burgess, "How satellite imagery magnified ukraine to the world," Apr 2024. [Online]. Available: https://www.wired.com/story/the-dangerous-rise-of-gps-attacks/

[5] L. Mathews, "Viasat reveals how russian hackers knocked thousands of ukrainians offline," 2022. [Online]. Available: https://www.forbes.com/sites/leemathews/2022/03/31/viasat-reveals-how-russian-hackers-knocked-thousands-of-ukrainians-offline/

[6] L. Whiting, "Viasat targeted in china-linked salt typhoon cyber campaign," Sep 2025. [Online]. Available: https://bastille.net/viasat-china-linked-salt-typhoon-cyber-campaign/

[7] S. Erwin, "Russia, china target spacex's starlink in escalating space electronic warfare," Apr 2025. [Online]. Available: https://spacenews.com/russia-china-target-spacexs-starlink-in-escalating-space-electronic-warfare/

[8] T. Pultarova, "White hat hackers expose iridium satellite security flaws," Jun 2025. [Online]. Available: https://spectrum.ieee.org/iridium-satellite

[9] S. Jero, J. Furgala, M. A. Heller, B. Nahill, S. Mergendahl, and R. Skowyra, "Securing the satellite software stack," in *Proceedings 2024 Workshop on Security of Space and Satellite Systems, San Diego, CA, USA: Internet Society*, 2024.

[10] J. Willbold, M. Schloegel, M. Vögele, M. Gerhardt, T. Holz, and A. Abbasi, "Space odyssey: An experimental software security analysis of satellites," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1–19.

[11] S. Havermans, L. Baumgärtner, J. Roberts, M. Wallum, and J. Caballero, "Fuzzing space communication protocols," in *Workshop on the Security of Space and Satellite Systems (SpaceSec)*, 2025.

[12] T. Scharnowski, F. Buchmann, S. Wörner, and T. Holz, "A case study on fuzzing satellite firmware," in *Workshop on the Security of Space and Satellite Systems (SpaceSec)*, 2023.

[13] A. Olchawa, M. Starcik, R. Fradique, and A. Boulaich, "Burning, trashing, spacecraft crashing: A collection of vulnerabilities that will end your space mission," in *BlackHat*, 2025. [Online]. Available: https://i.blackhat.com/BH-USA-25/Presentations/USA-25-OlchawaStarcik-Burning-Trashing-Spacecraft-Crashing.pdf

[14] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, pp. 1–33, 2017.

[15] N. Burow, X. Zhang, and M. Payer, "Sok: Shining light on shadow stacks," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 985–999.

[16] H. Lefeuvre, N. Dautenhahn, D. Chisnall, and P. Olivier, " SoK: Software Compartmentalization ," in *2025 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00075

[17] D. P. McKee, Y. Giannaris, C. Ortega, H. E. Shrobe, M. Payer, H. Okhravi, and N. Burow, "Preventing kernel hacks with hakcs." in *NDSS*, 2022, pp. 1–17.

[18] A. Koprowski and H. Binsztok, "Trx: A formally verified parser interpreter," *Logical Methods in Computer Science*, vol. 7, 2011.

[19] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an os microkernel," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 1, pp. 1–70, 2014.

[20] core Flight System Team, "core flight system," 2023. [Online]. Available: https://cfs.gsfc.nasa.gov/

[21] S. Jero, J. Furgala, R. Pan, P. K. Gadepalli, A. Clifford, B. Ye, R. Khazan, B. C. Ward, G. Parmer, and R. Skowyra, "Practical principle of least privilege for secure embedded systems," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2021, pp. 1–13.

[22] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[23] D. Werner, "Nasa to roll out major update to core flight software," Feb 2025. [Online]. Available: https://spacenews.com/nasa-to-roll-out-major-update-to-core-flight-software/

[24] NASA Goddard, "Celebrating twenty years of core flight software," 2024. [Online]. Available: https://etd.gsfc.nasa.gov/capabilities/core-flight-system/news/celebrating-twenty-years-of-core-flight-software/

[25] E. Geist, "Core flight system (cfs) training," NASA, Tech. Rep. 20240000217, Jan 2024. [Online]. Available: https://ntrs.nasa.gov/citations/20240000217

[26] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 207–220.

[27] G. Heiser and K. Elphinstone, "L4 microkernels: The lessons from 20 years of research and deployment," *ACM Transactions on Computer Systems (TOCS)*, vol. 34, no. 1, pp. 1–29, 2016.

[28] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, 1966.

[29] I. Kuz, Y. Liu, I. Gorton, and G. Heiser, "Camkes: A component model for secure microkernel-based embedded systems," *Journal of Systems and Software*, vol. 80, no. 5, pp. 687–699, 2007.

[30] The seL4 Foundation, "Microkit user manual," 2025. [Online]. Available: https://github.com/seL4/microkit/blob/main/docs/manual.md

[31] R. W. Skowyra, S. A. Mergendahl, and R. Khazan, "Holding the high ground: Defending satellites from cyber attack," 2023. [Online]. Available: https://www.afcea.org/signal-media/cyber-edge/holding-high-ground-defending-satellites-cyber-attack

[32] F Prime Team, "F prime: A software ecosystem enabling the rapid development and deployment of embedded systems for spaceflight applications," 2025. [Online]. Available: https://fprime.jpl.nasa.gov/

[33] L. Tung, "Spacex: We've launched 32,000 linux computers into space for starlink internet," 2020. [Online]. Available: https://www.zdnet.com/article/spacex-weve-launched-32000-linux-computers-into-space-for-starlink-internet/

[34] B. C. Ward, R. Skowyra, S. Jero, N. Burow, H. Okhravi, H. Shrobe, and R. Khazan, "Security considerations for next-generation operating systems for cyber-physical systems," in *1st International Workshop on Next-Generation Operating Systems for Cyber-Physical Systems (NGOSCPS)*, 2019.

[35] The RTEMS Project, "The rtems project," Jan 2026. [Online]. Available: https://rtems.org/

[36] Wind River Software, "Vxworks: Real-time os for mission-critical systems," Jan 2026. [Online]. Available: https://www.windriver.com/products/embedded/vxworks

[37] The RTEMS Project, "Rtems: Memory management manager," Jan 2026. [Online]. Available: https://docs.rtems.org/docs/6.2/posix-users/memory_managment.html