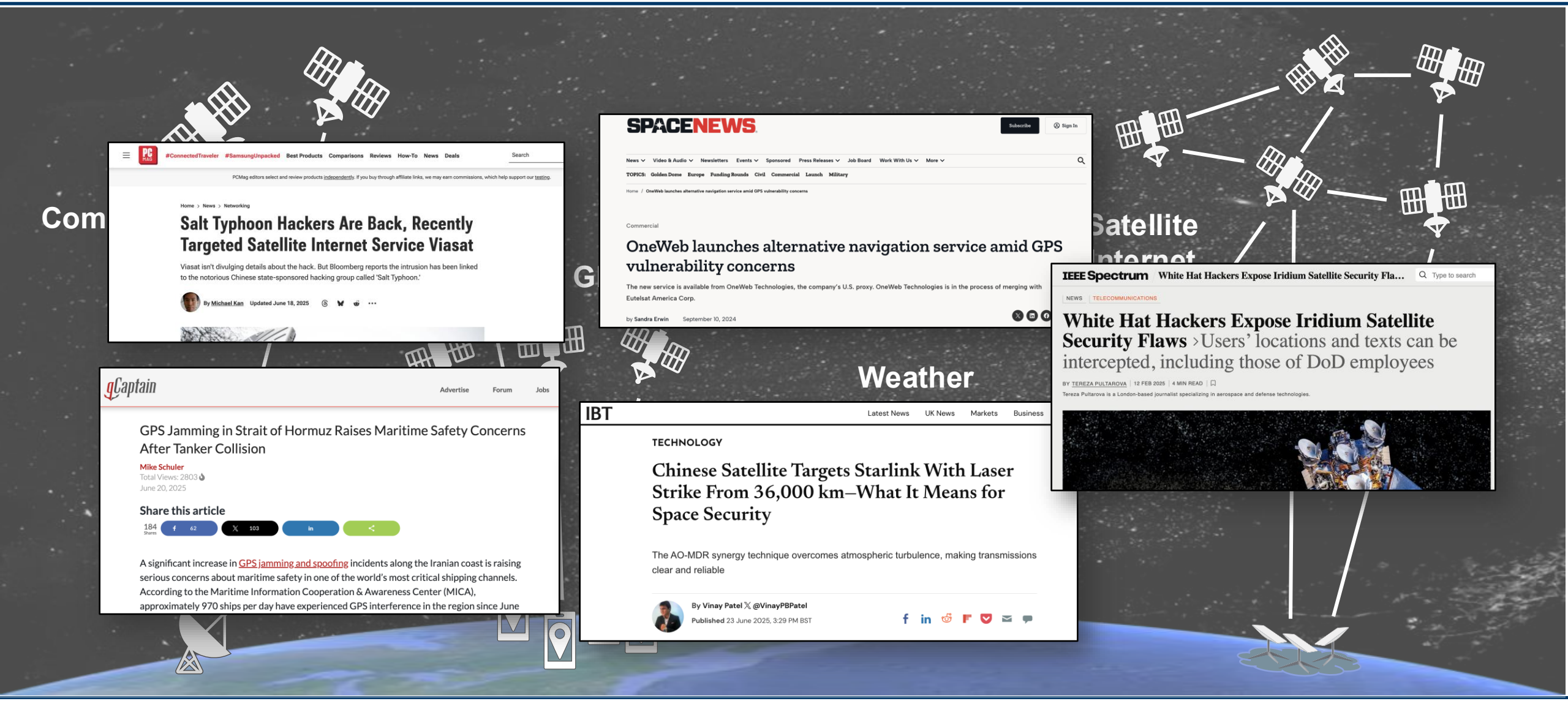
Porting NASA core Flight System to Magnetite on seL4

Juliana Furgala

September 2025

LINCOLN LABORATORY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

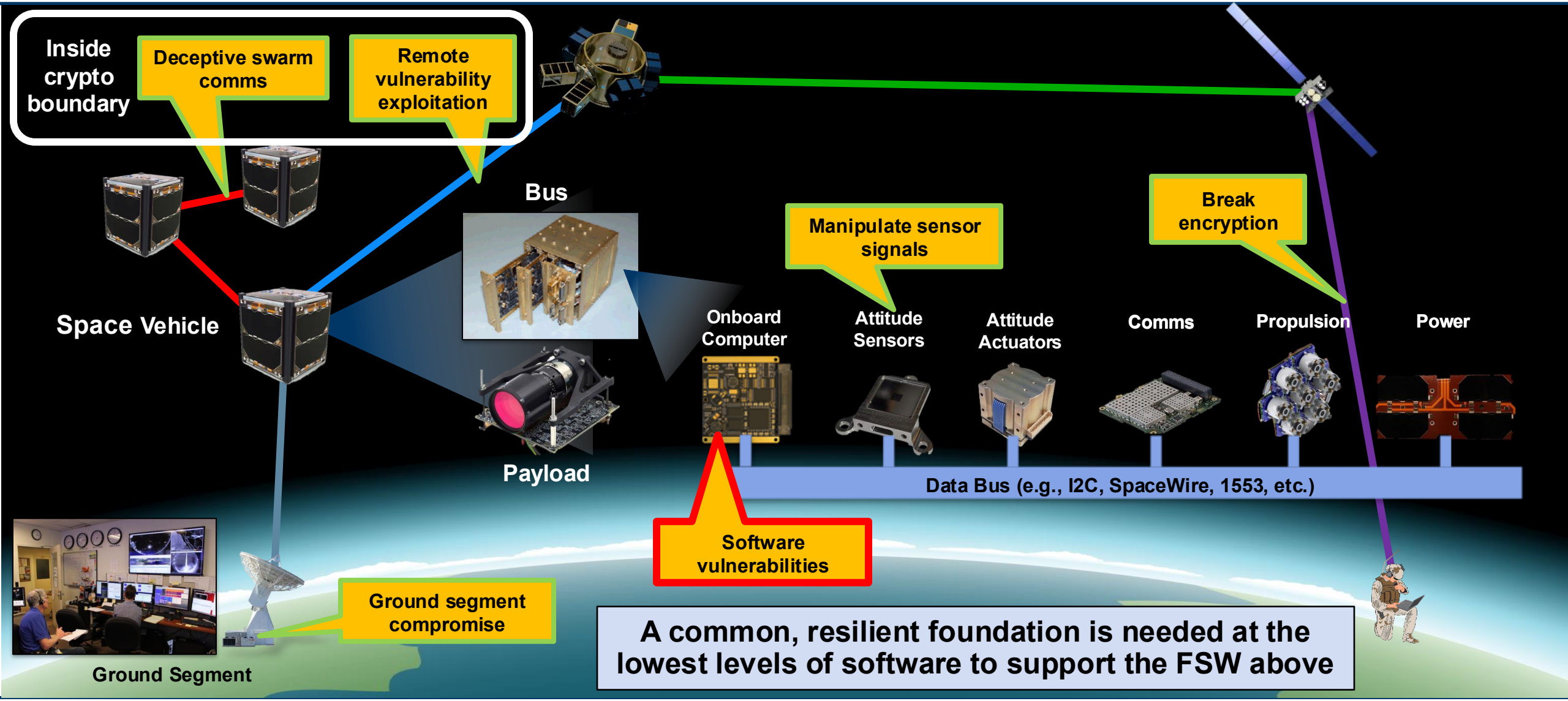
Satellite Systems are Valuable



LEO: Low Earth Orbit

Space-Cyber Threat Vectors

Has existing protections or is out of scope

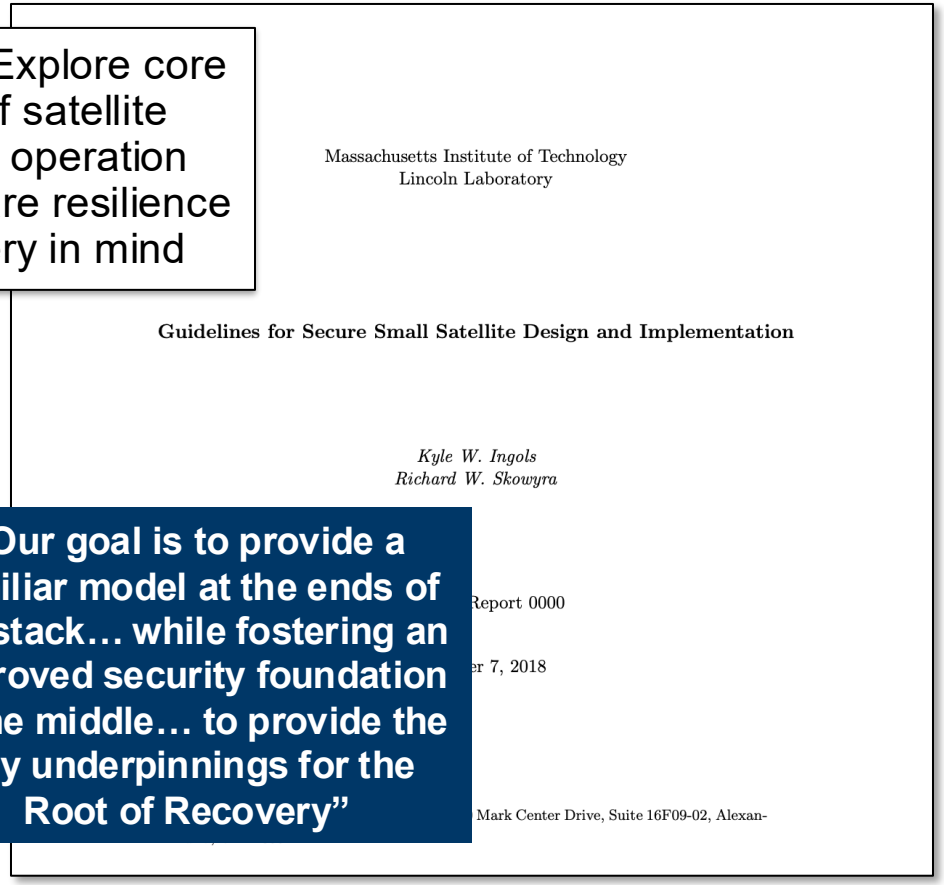


A common, resilient foundation is needed at the lowest levels of software to support the FSW above

FSW: Flight Software
SWaP: Size, Weight, and Power

SmallSat Guidelines Report

Purpose: Explore core elements of satellite design and operation with software resilience and recovery in mind



Massachusetts Institute of Technology
Lincoln Laboratory

Guidelines for Secure Small Satellite Design and Implementation

Kyle W. Ingols
Richard W. Skowyra

Report 0000

er 7, 2018

Mark Center Drive, Suite 16F09-02, Alexan-

"Our goal is to provide a familiar model at the ends of the stack… while fostering an improved security foundation in the middle… to provide the key underpinnings for the Root of Recovery"

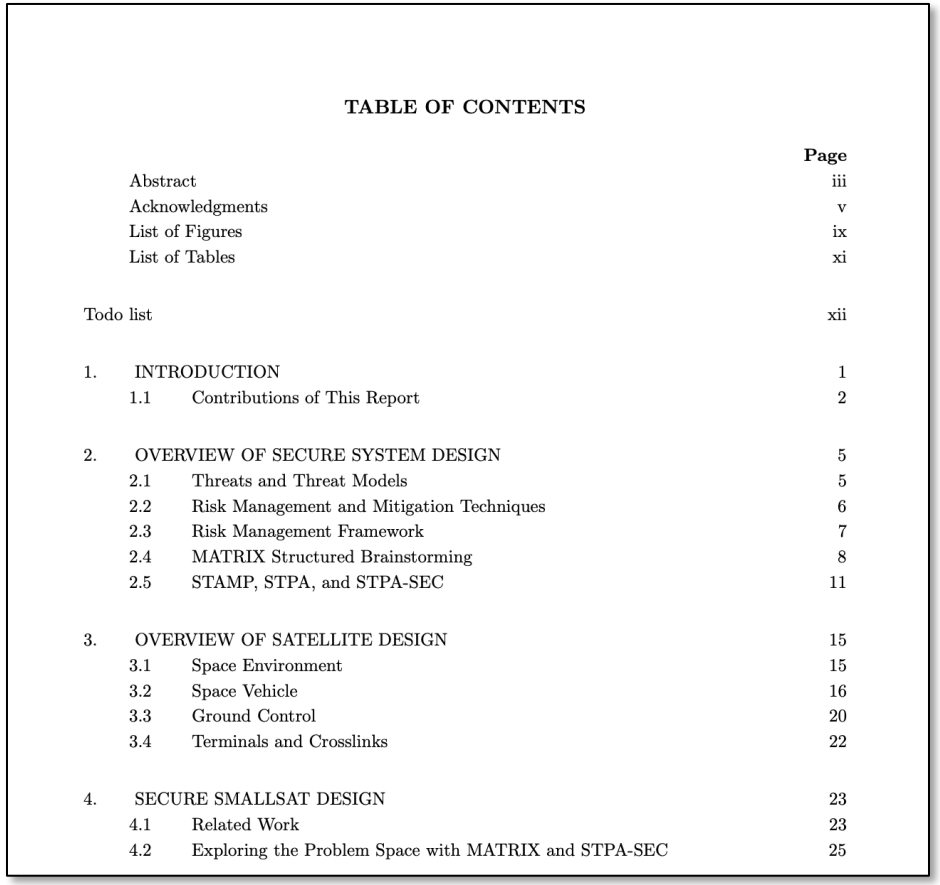
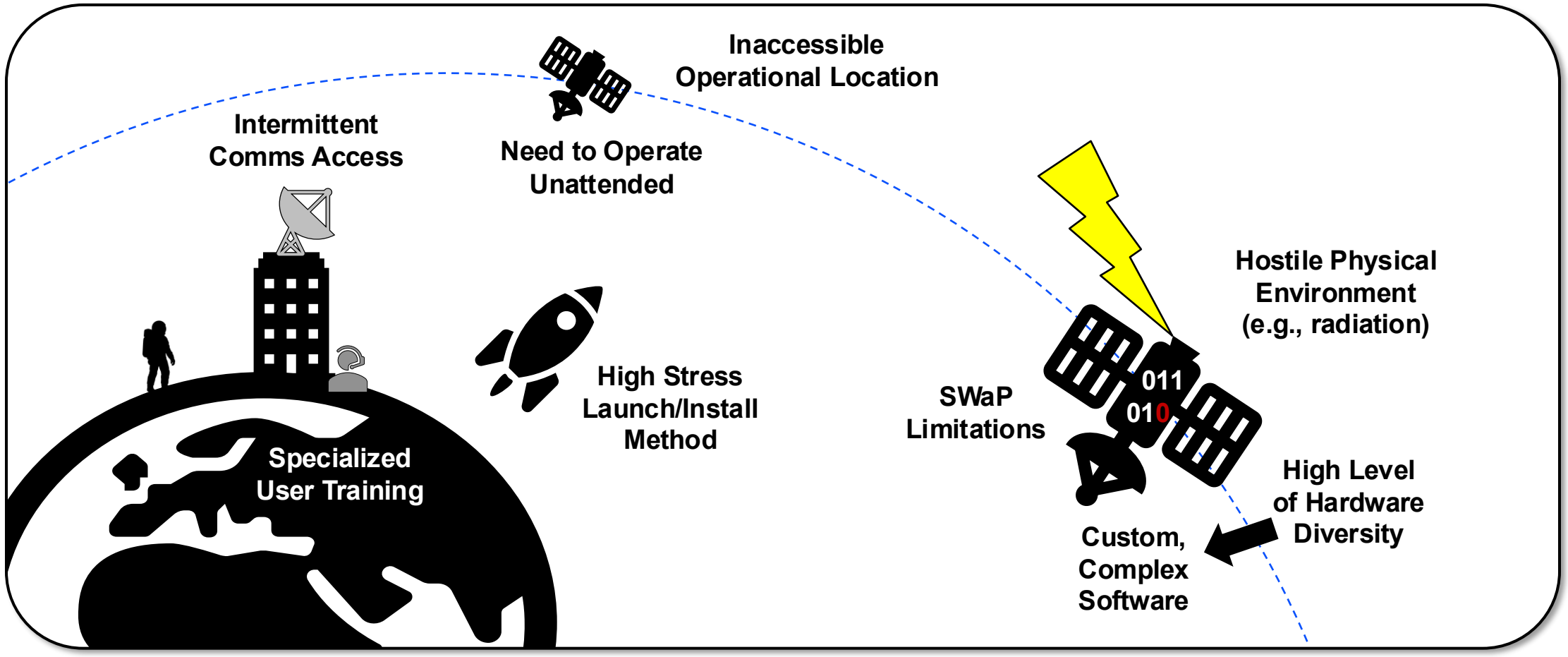


TABLE OF CONTENTS

Flight software needs a secure foundation, but requires a systematic approach to develop

https://www.ll.mit.edu/sites/default/files/publication/doc/guidelines-secure-small-satellite-design-ingols-lsp-249.pdf

Challenges of Space System Survival



Not to scale

LEO: Low Earth Orbit
SWaP: Size Weight and Power

seL4

- **Formally verified microkernel**

Functional Correctness

Free From Memory Bugs

Binary Correctness

Data Integrity

Controlled Information Flow

2

Comprehensive Formal Verification of an OS Microkernel

GERWIN KLEIN, JUNE ANDRONICK, KEVIN ELPHINSTONE, TOBY MURRAY, THOMAS SEWELL, RAFAL KOLANSKI, and GERNOT HEISER, NICTA and UNSW, Sydney, Australia

We present an in-depth coverage of the comprehensive machine-checked formal verification of seL4, a general-purpose operating system microkernel.

We discuss the kernel design we used to make its verification tractable. We then describe the functional correctness proof of the kernel's C implementation and we cover further steps that transform this result into a comprehensive formal verification of the kernel: a formally verified IPC fastpath, a proof that the binary code of the kernel correctly implements the C semantics, a proof of correct access-control enforcement, a proof of information-flow noninterference, a sound worst-case execution time analysis of the binary, and an automatic initialiser for user-level systems that connects kernel-level access-control enforcement with reasoning about system behaviour. We summarise these results and show how they integrate to form a coherent overall analysis, backed by machine-checked, end-to-end theorems.

The seL4 microkernel is currently not just the only general-purpose operating system kernel that is fully formally verified to this degree. It is also the only example of formal proof of this scale that is kept current as the requirements, design and implementation of the system evolve over almost a decade. We report on our experience in maintaining this evolving formally verified code base.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; D.4.5 [**Operating Systems**]: Reliability—*Verification*

General Terms: Verification; Security; Reliability

Additional Key Words and Phrases: seL4, Isabelle/HOL, operating systems, microkernel, L4

ACM Reference Format:
Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.* 32, 1, Article 2 (February 2014), 70 pages.
DOI: http://dx.doi.org/10.1145/2560537

1. INTRODUCTION

This article presents a detailed coverage of the comprehensive formal verification of the seL4 microkernel, from its initial functional correctness proof to more recent results, which extend the assurance argument up to higher-level security properties and down to the binary level of its implementation.

The target of our verification, the kernel, is the most critical part of a system, which is our motivation for starting system verification with this component. The customary definition of a kernel is the software that executes in the privileged mode of the hard-

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.
Authors' address: NICTA, Level 4, 223 Anzac Pde, Sydney NSW 2052, Australia; Correspondence email: gernot@nicta.com.au.

DOI: http://dx.doi.org/10.1145/2560537

- **Has scheduling, capabilities, and IPC**
- **Does NOT have system services, drivers, sense of processes, etc.**

**seL4 provides capabilities and performant IPC,
upon which we build a set of independent system services**

Our Approach

NASA core Flight System

Synchronization Service	Timer Service	Logging Service	Channel Service
Ethernet Service	Network Service	Event Service	Threads & Processes

Formally verified microkernel

Hardware

MAGNETITE

Xilinx Zynq702

Outline

- Motivation
- Mission Application: cFS
- Porting cFS to seL4
- Evaluation
- Lessons Learned

Purpose of Flight Software

Radio Frequency

Command & Data Handling

Power Control

Navigation & Control

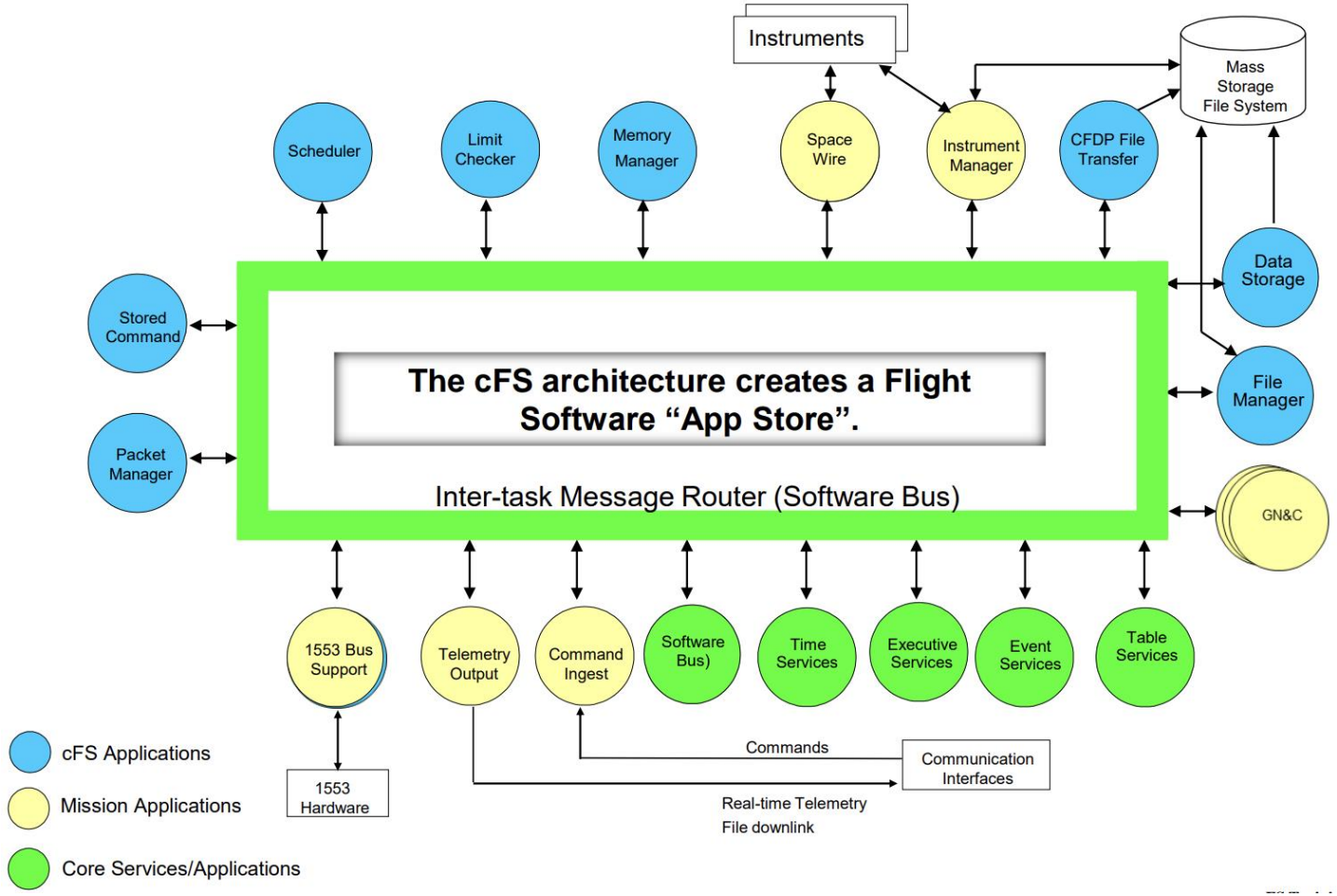
Guidance

Instrument/ Sensor Handling

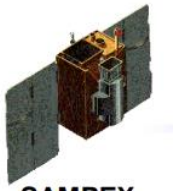
Flight software is the (ideally resilient) real-time “brain” that controls mission operations

NASA's core Flight System (cFS)

- **Widely used flight software framework and application suite**
- **Canonical open-source choice for academic research & government use**
- **Designed as a reusable application layer for space system command and control, providing navigation, guidance, etc. in common modules**



NASA cFS' Heritage Story

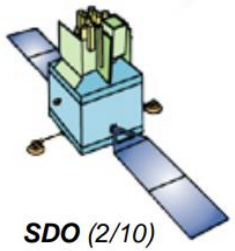


SAMPEX
(8/92)

1990s

"Clone and own" software development

2005



SDO (2/10)

core Flight Executive (cFE) developed

2007

LRO
(2009 launch)

2010

Initial cFS applications developed

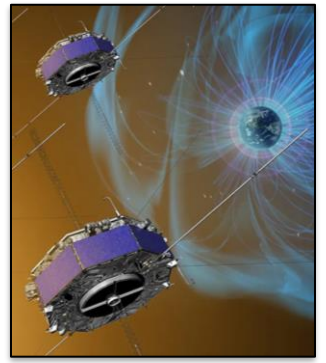
cFE open-sourced

2011

LADEE launch

GPM launch

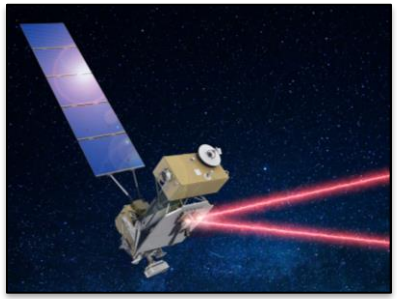
2014



MMS launch

cFS open sourcing planned

2016

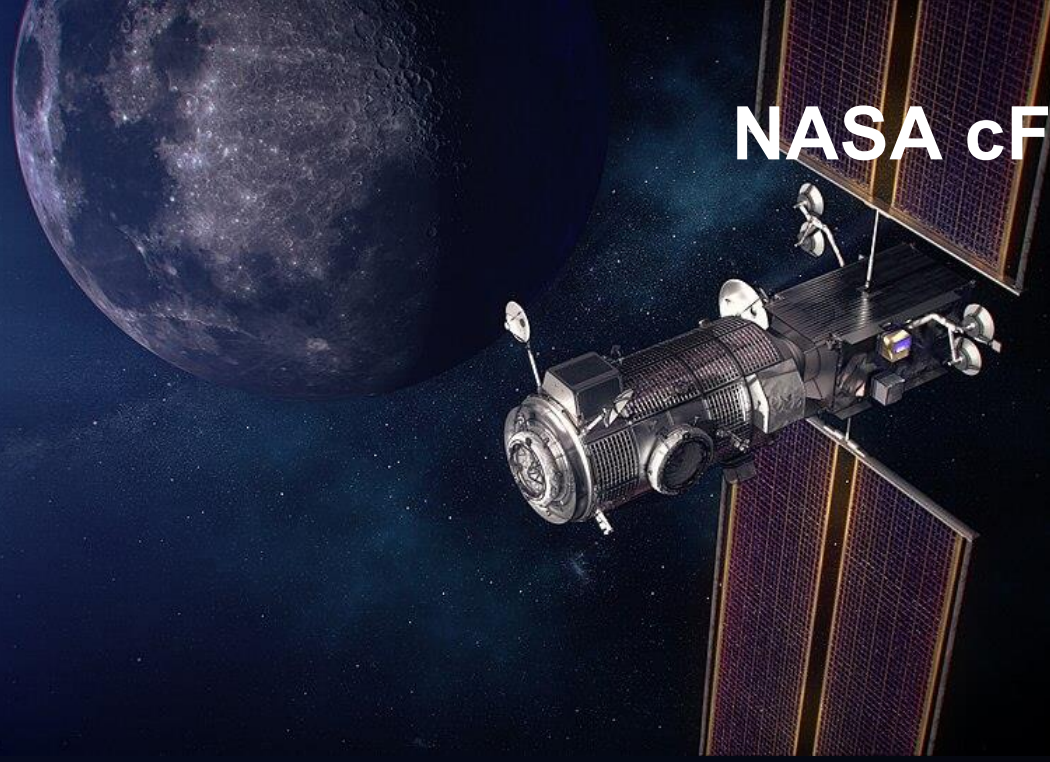


LCRD launch

GPM: Global Precipitation Measurement
LADEE: Lunar Atmosphere and Dust Environment Explorer
LCRD: Laser Communications Relay Demonstration

LRO: Lunar Reconnaissance Orbiter
MMS: Magnetospheric Multiscale
SDO: Solar Dynamic Observatory

NASA cFS' Heritage Story

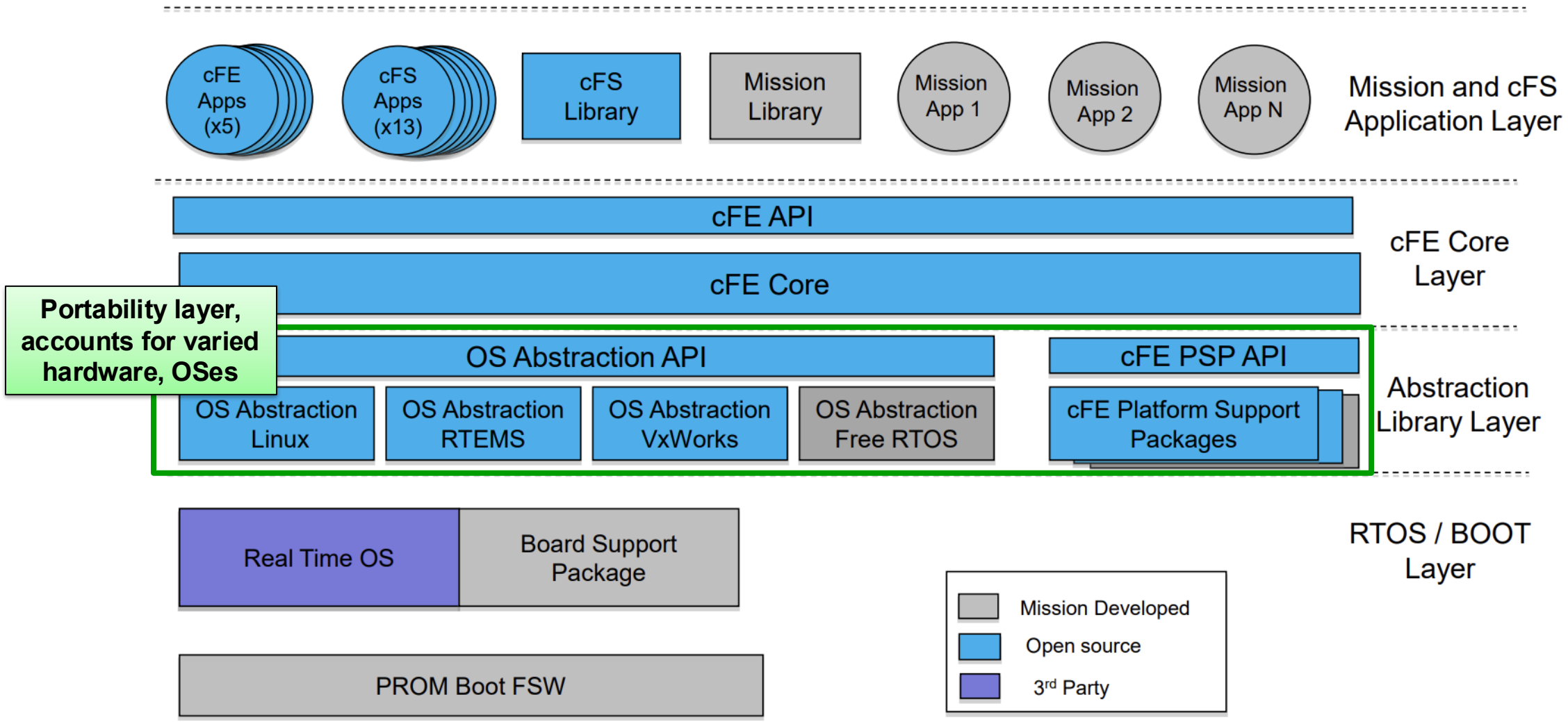


Artemis

Used on more than 40 projects, including landers, orbiters, unnamed aerial vehicles, space suits, crew habitats, rovers, satellites

Roman Space Telescope

cFS Design and Structure



FSW: Flight Software
OS: Operating System
PSP: Platform Support Package

cFS Design Observations

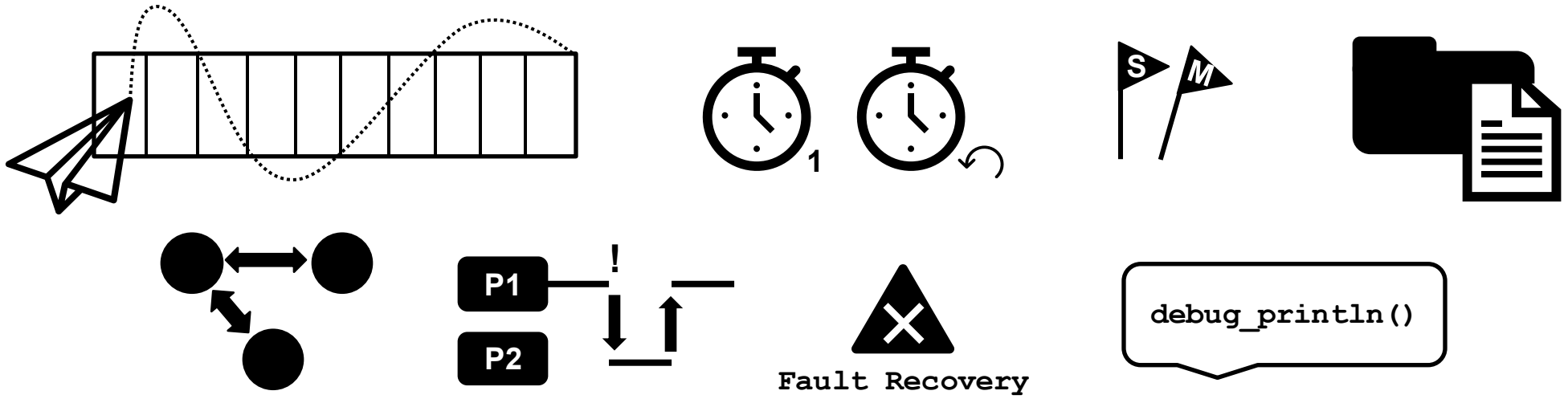
- No usage of heap memory
  - Common in real time applications to help with predictability
  - Instead MANY global variables are used
- cFS is one address space with many threads
  - Each app has at least one thread
  - Extensive use of memory spaces that are shared between threads
- Apps are dynamic
  - Expect to be able to start and stop components
  - Can be stopped and started at runtime (by other components or the ground)
  - Can be added and removed at runtime (using dynamic libraries)
- Availability is an overarching priority
  - Mutexes are reentrant
  - Many operations have timeouts

Outline

- Motivation
- Mission Application: cFS
- Porting cFS to seL4
- Evaluation
- Lessons Learned

Analysis of NASA's cFS

- Applications rely on an OS Abstraction Layer, which then calls the underlying OS functionality
- This eases porting, as OS-specific functionality is implemented in only one place
- OS Abstraction Layer consists of about 100 API calls
- Functionality Expected:

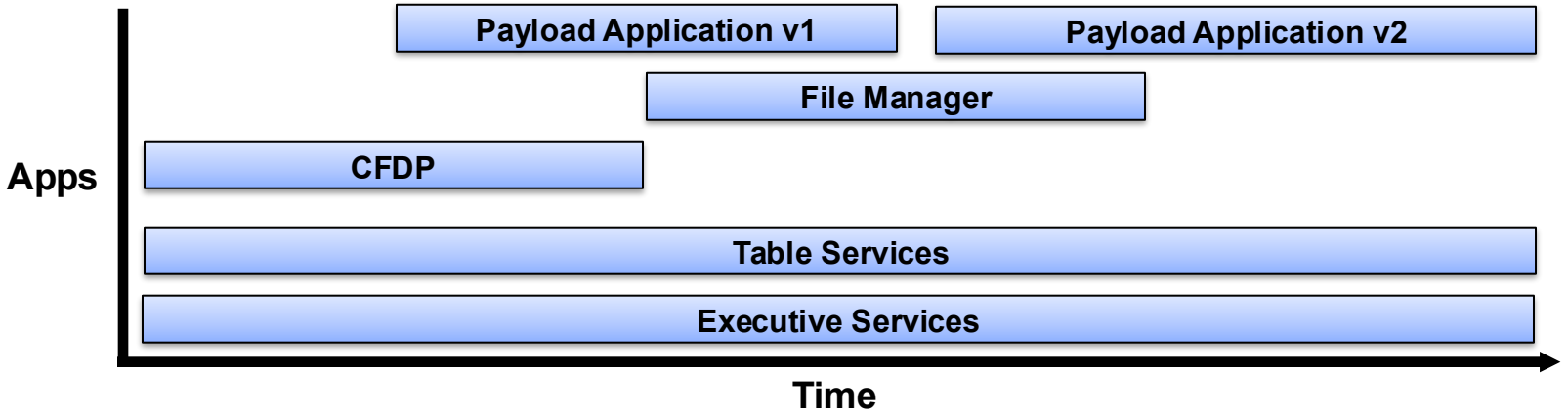


cFS's OS Abstraction Layer requires significant functionality from an underlying system

API: Application Program Interface

Dynamism Mismatch

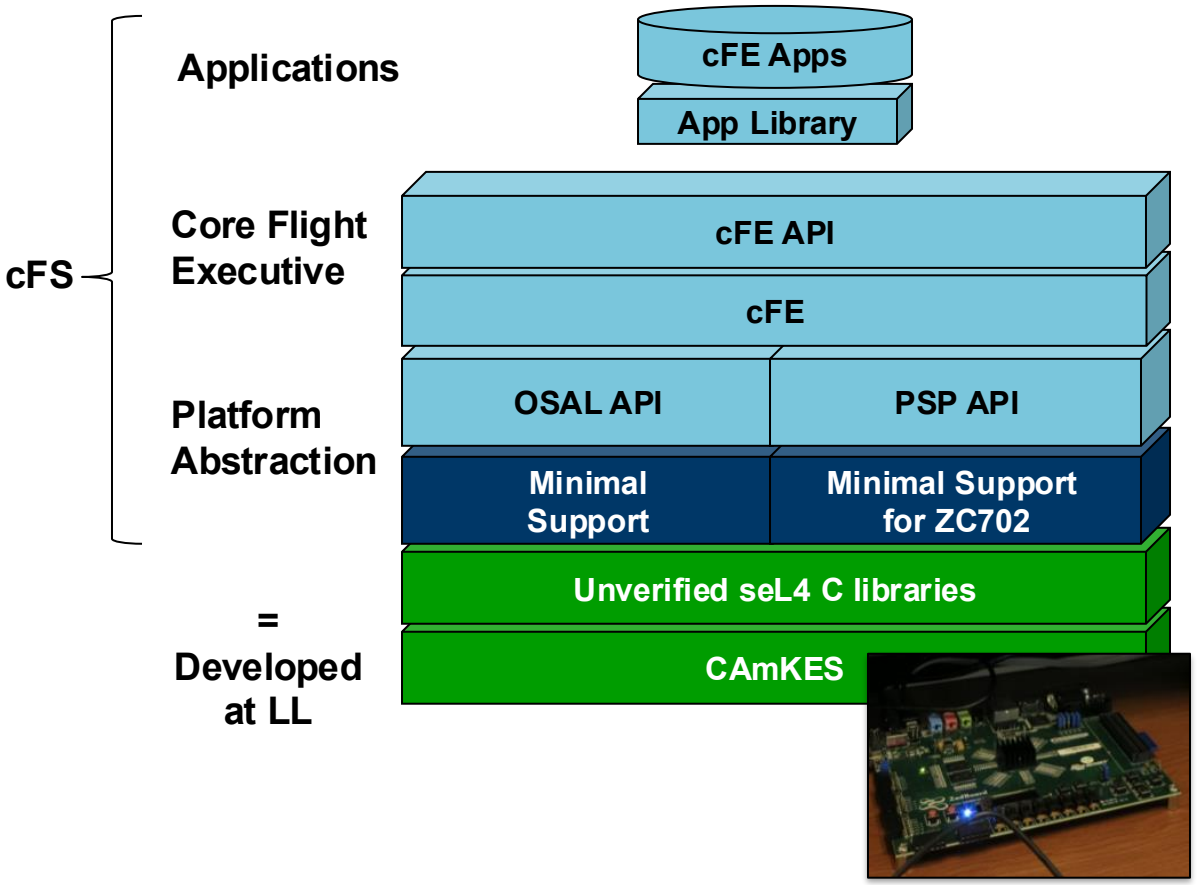
- cFS is surprisingly dynamic
  - Apps can be started and stopped
  - Apps can be added and removed at runtime
- Resources required by the system change at runtime
  - Threads, mutexes, semaphores, channels, timers, memory
- Much prior work on seL4 assumes static resource allocation
  - CAmKES, Microkit, others
- Initial Solution: Dynamically instrument cFS to find a typical upper bound on resource usage
- Final Solution: Create a solution supporting dynamic creation of resources



Initial Proof of Concept

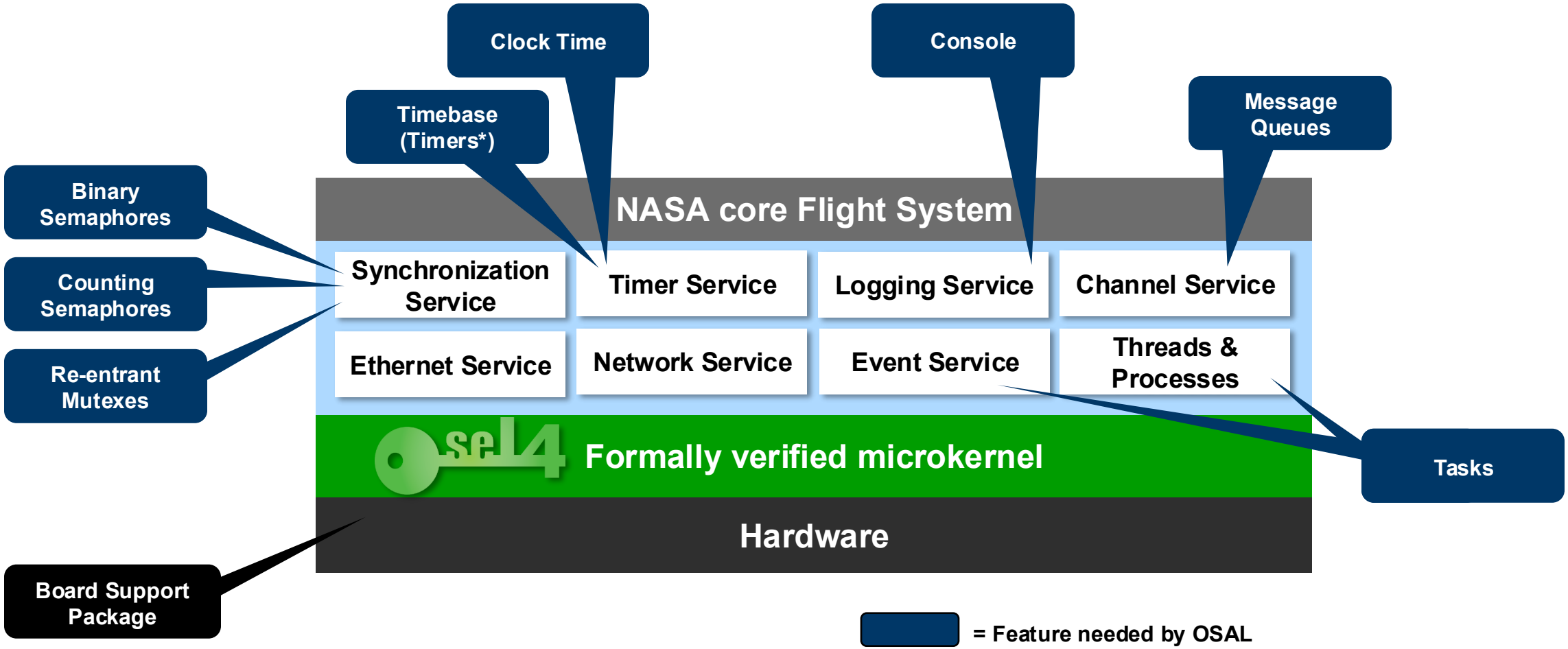
- Built on CAmKES and unverified seL4 C libraries
- Hardcoded the apps that started, preventing the runtime startup of apps
- Stubbed out OSAL APIs where possible
- Required features leaned on C libraries
- Particular difficulty with semaphores, mutexes, setting estimated ceiling on resources

Difficulty of resource management and lack of dynamism motivates the need for an actual OS



API: Application Program Interface
OSAL: Operating System Abstraction Layer
PSP: Platform Support Package

Designing an OS for cFS



Magnetite is the Operating System that resulted from this process

OSAL: Operating System Abstraction Layer

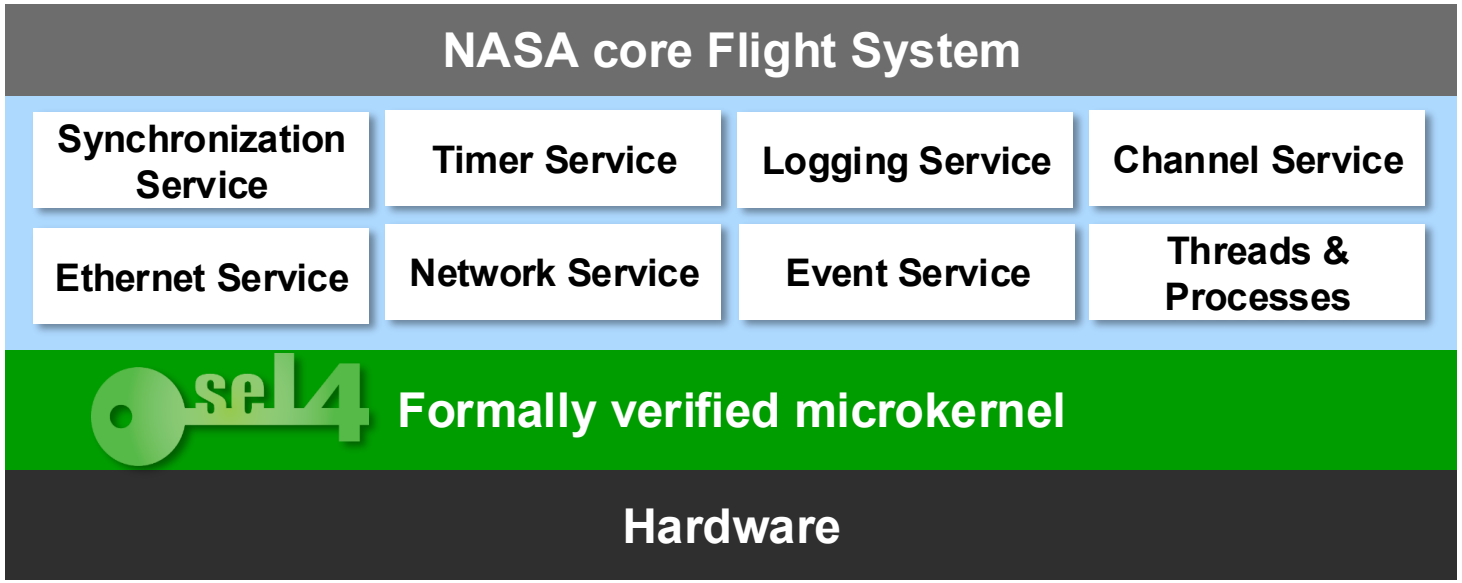
* One-shot and recurring

Secure Design Principles

Decentralization of responsibility

Principle of least privilege

Built upon formal methods foundation



Security-first principles informed initial system service design and that of our resulting OS

OSAL: Operating System Abstraction Layer

* One-shot and recurring

Magnetite Development Timeline

cFS deployed on dozens of space missions

2018

SmallSat Guidelines Research

2019

seL4 building, capability and IPC handling

2020

ELF loading, cFS integration

2021

First NASA cFS Demo

Silos introduced

2022

v1

Zynq702

ARMv7

2023

v2-v3

ARMv7

DM3730

2024

v4-v6

ODROID-H3

x86-64

TX2

ARMv8

v7

ZCU102

ARMv8

2025

v8

RISC-V

PolarFire

ARMv8

VCK190

v9

ARMv8

Kria K24

Services Developed

v1: timer, channel, event, sync

v2-3: shared memory, signal, UART logging, NVM, principles

v4: file system

v6: I2C

v7: cross-silo networking, ticker

v8: boot

ELF: Executable and Linkable Format
IPC: Inter-Process Communication
NVM: Non-Volatile Memory

= version release

Performance Microbenchmarks

Overhead Measurements (Cycles)

	Real-Time Patched Linux				Magnetite (2021)				Magnetite (2022)			
	Average	Std Dev	95 %tile	Max	Average	Std Dev	95 %tile	Max	Average	Std Dev	95 %tile	Max
Context Switch: Thread	1,060	25	1,077	3,232	542	12	563	597	504	0	504	550
Context Switch: Process	4,816	327	4,858	17,919	542	12	564	703	498	1	498	599
Round Trip IPC	*	*	*	*	989	19	1,027	1,113	1,136	3	1,137	1,241
Event Latency: equal prio	*	*	*	*	11,504	175	11,801	12,247	8,788	185	9,095	10,393
Event Latency: L2H prio	*	*	*	*	11,407	176	11,702	12,233	8,790	181	9,093	9,870
Event Latency: H2L prio	*	*	*	*	16,585	222	16,953	18,160	14,138	292	14,613	17,614
Mutex Uncontended	217	2	217	328	9,959	184	10,270	11,165	6,301	292	6,745	8,615
Mutex Contended	15,844	619	16,263	30,570	13,053	234	13,440	13,918	15,574	285	16,042	17,394
Semaphore Uncontended	116	90	116	9,112	9,051	179	9,357	9,792	5,360	200	5,689	6,348
Semaphore Contended	6,713	404	6,994	22,136	11,430	217	11,791	12,384	11,661	250	12,070	12,741
Timer Latency	20,665	1,068	21,171	33,118	16,042	203	16,381	17,317	12,202	210	12,536	13,907
Timer Latency w/ timerfd	6,493	632	6,842	14,806								
Channel Latency: L2H prio	9,439	423	9,627	22,671	23,749	230	24,138	25,678	18,367	286	18,850	20,038
Channel Latency: H2L prio	11,507	841	11,711	71,169	24,839	229	25,222	27,806	18,505	273	18,983	20,271

Microbenchmarks show resource primitives to be performant, with improvement over time

CDF: Cumulative Distribution Functions
IPC: Inter-Process Communication

* No direct Linux equivalent

Outline

- Motivation
- Mission Application: cFS
- Porting cFS to an OS on seL4
- Evaluation
- Lessons Learned

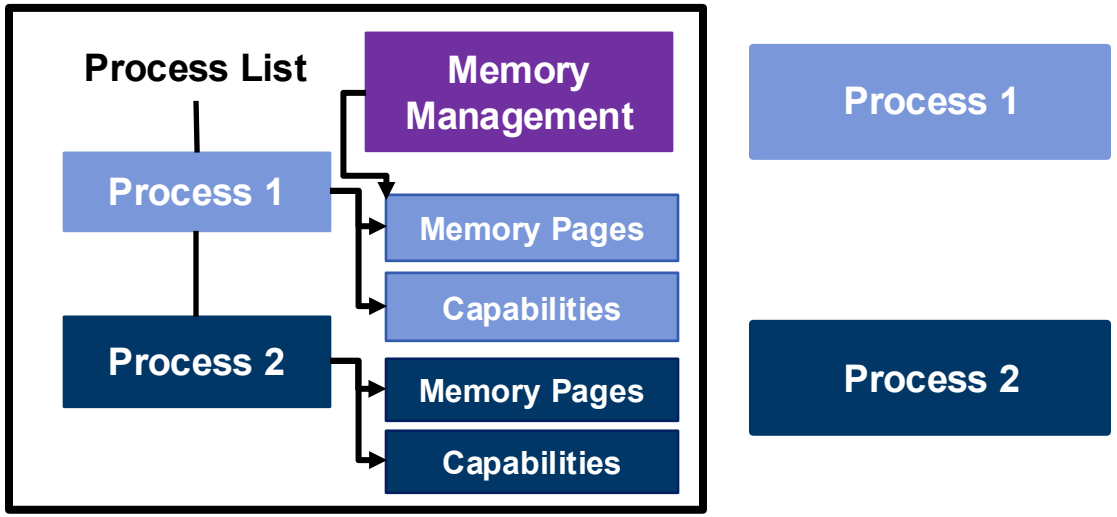
Lessons Learned: Space Software is Dynamic

- Space Flight Software has significant dynamism
  - Apps can be started or stopped, added or removed from the system
  - Functionality can be enabled or disabled at runtime
  - Partial updates are common
- This leads to changes in system configuration and resource usage at runtime
- Static system configurations are inadequate
  - Would require many system images which is complex to create
  - Low bitrates mean sending those images is expensive
  - Updates are considered risky in space

Space software requires an operating system that can create new dynamic system configurations and resources

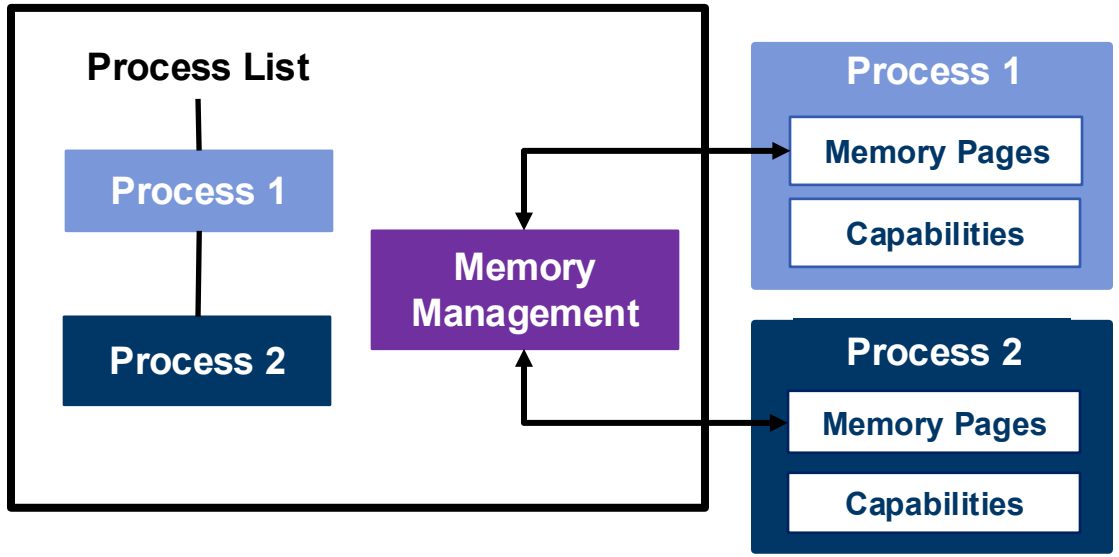
Lessons Learned: Minimizing Temporal Trust

Linux Processes



- Privileged system processes manage user process memory and capabilities
- User processes are captive in trust to the OS
- Privilege is centralized and compromise spreads

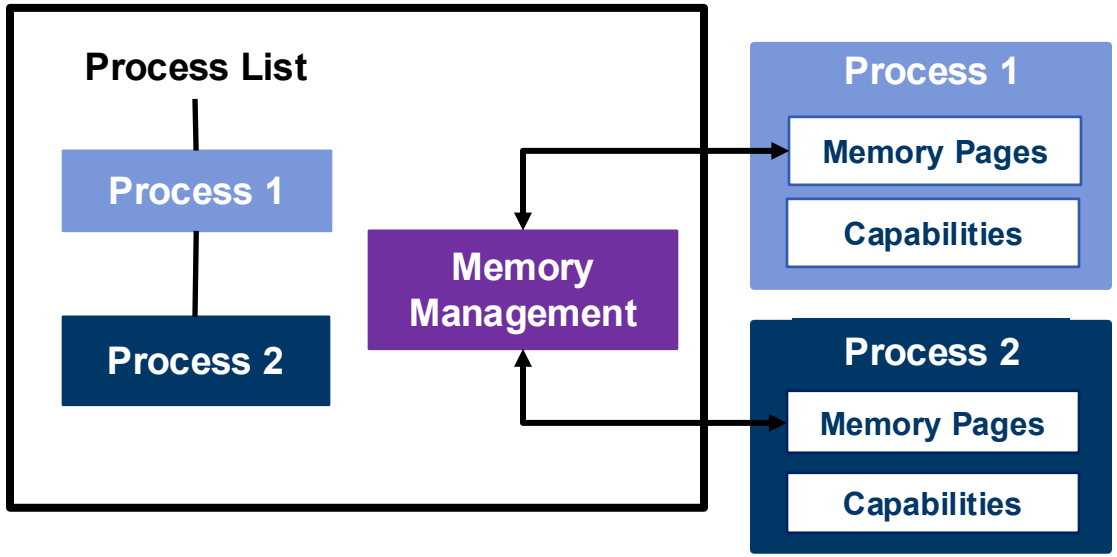
Self-Contained Processes



- Self-owned memory and capabilities
- System services, like memory management, only *borrow* needed capabilities
- Intentional decentralization of privilege
- System services are only trusted at use time

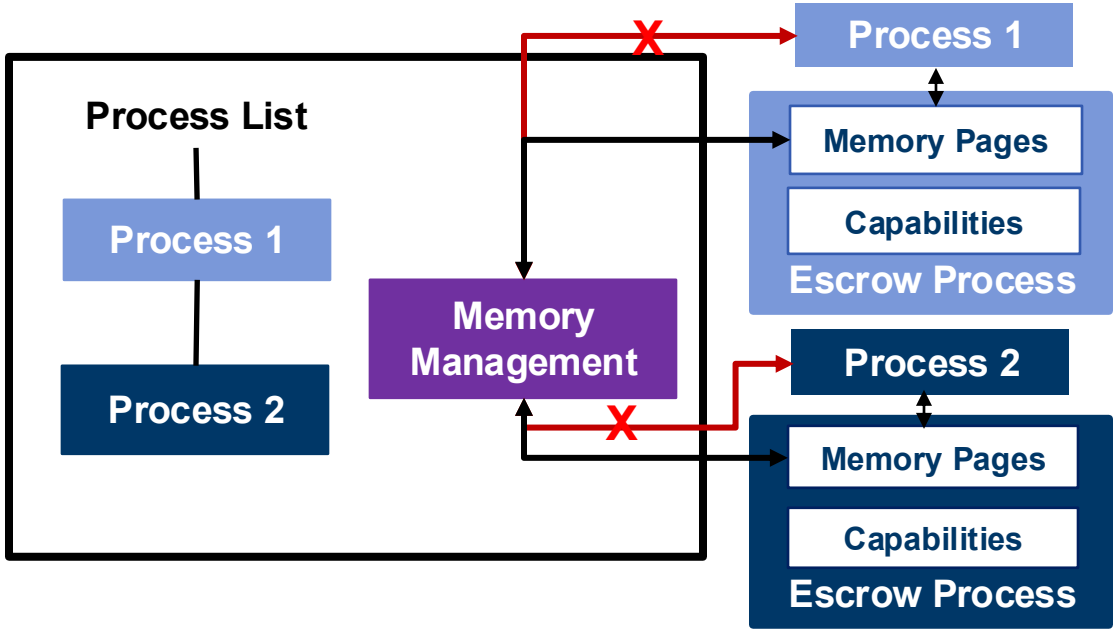
Lessons Learned: Minimizing Temporal Trust

Self-Contained Processes



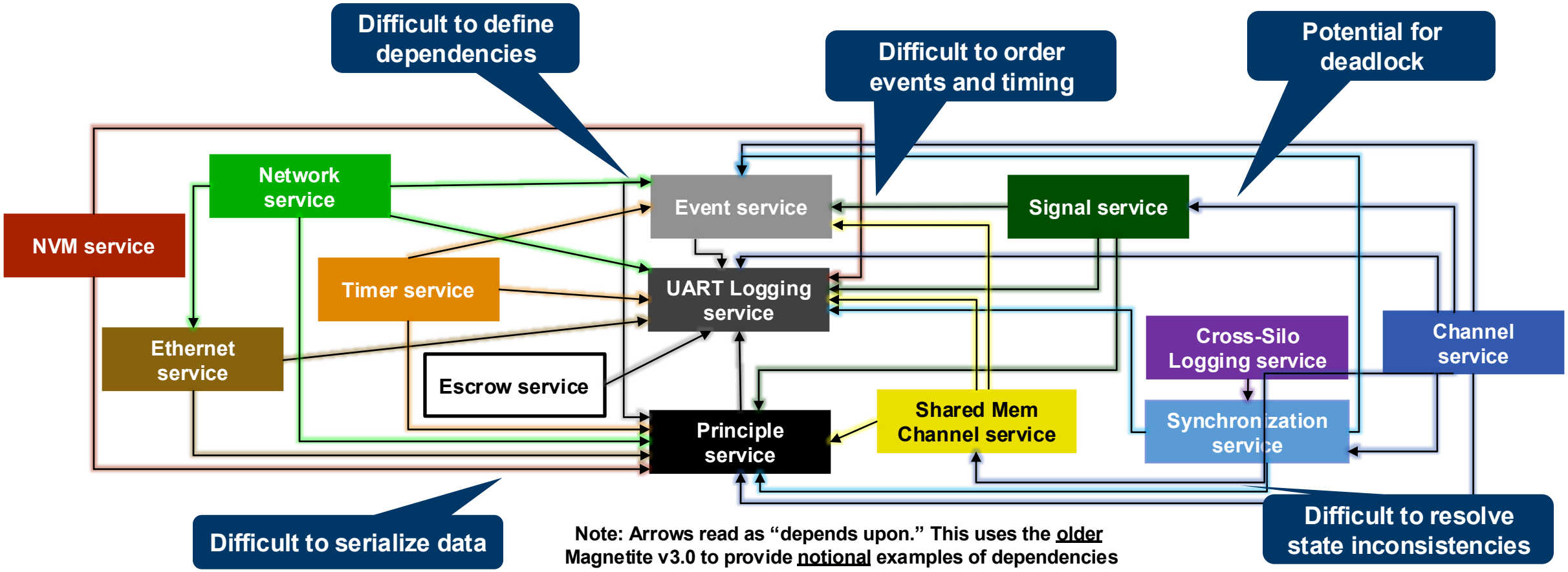
- Issue: Processes can rearrange their cspace to prevent memory reclamation
- Deleting a cspace does not delete ones within it, making them unreachable
- Reidentifying a capability is a storage side channel

Escrow Processes



- Hold process capabilities "in escrow"
- User process → system service requests that use cspace capabilities pass through an escrow process
- Minimal trust needed between user processes and services (and vice versa)

From an Operating System to a Distributed System



Note: Arrows read as "depends upon." This uses the older Magnetite v3.0 to provide notional examples of dependencies

A microkernel approach with many services brings with it all the classic problems from distributed systems

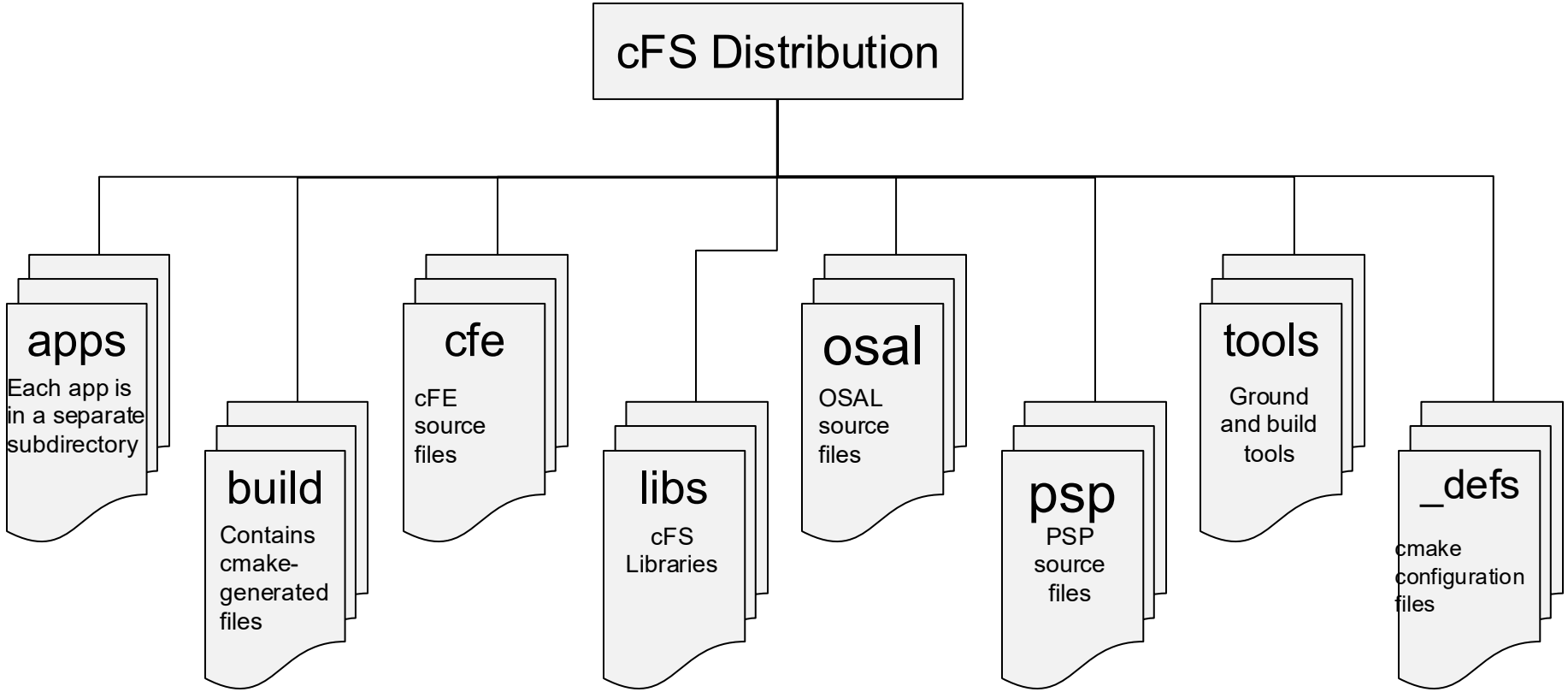
Conclusion

- Flight software is critical in bringing satellites to life and ensuring they stay in an operational state during their missions
- Decades of space excursions have relied upon very stovepiped, tightly coupled software-hardware solutions
- However, with space being an increasingly accessible operating environment and therefore a tantalizing target, flight software needs to run on a secure foundation
- cFS, a more modularly designed modern FSW solution, was selected and we studied the minimum resources it needed to operate
- Through the process of porting cFS to seL4 (really Magnetite OS), we learned lessons about the surprising dynamism of space FSW, how to minimize trust, and the difficulties of distributed systems as applied to OSes

Contact Information

Juliana Furgala
Juliana.Furgala@ll.mit.edu

cFS Mission Directory Structure



GSFC Open Source Apps

Application	Function
CFDP	Transfers/receives file data to/from the ground
Checksum	Performs data integrity checking of memory, tables and files
Command Ingest Lab	Accepts CCSDS telecommand packets over a UDP/IP port
Data Storage	Records housekeeping, engineering and science data onboard for downlink
File Manager	Interfaces to the ground for managing files
Housekeeping	Collects and re-packages telemetry from other applications.
Health and Safety	Ensures critical tasks check-in, services watchdog, detects CPU hogging, calculates CPU utilization
Limit Checker	Provides the capability to monitor values and take action when exceed threshold
Memory Dwell	Allows ground to telemeter the contents of memory locations. Useful for debugging
Memory Manager	Provides the ability to load and dump memory
Software Bus Network	Passes Software Bus messages over various “plug-in” network protocols
Scheduler	Schedules onboard activities via (e.g. HK requests)
Scheduler Lab	Simple activity scheduler with a one second resolution
Stored Command	Onboard Commands Sequencer (absolute and relative)
Stored Command Absolute	Allows concurrent processing of up to 5 (configurable) absolute time sequences
Telemetry Output Lab	Sends CCSDS telemetry packets over a UDP/IP port