ENGINEERING CYBER RESILIENCE IN SPACECRAFT FLIGHT SOFTWARE: A THREAT-INFORMED ARCHITECTURE AND EVALUATION

by
James R. Curbo

A dissertation submitted to Johns Hopkins University
in conformity with the requirements for
the degree of Doctor of Engineering

Baltimore, Maryland
May 2026

© 2026 James R. Curbo
All Rights Reserved

Abstract

Spacecraft flight software has traditionally been engineered for reliability, determinism, and fault tolerance, but not for sustained operation under cyber attack. As spacecraft become more software-driven, more interconnected, and more dependent on autonomous operation, that gap becomes a mission risk. Cyber resilience can be treated as a systems-engineering property of flight software: derived from threats, embedded in architecture, and evaluated through observable behavior.

A bottom-up attack surface analysis of a representative open-source flight software stack identifies inherited vulnerabilities and architectural conditions that allow compromise to propagate. A requirements workflow connects mission consequences, threat-informed analysis, and component-level “shall” statements so that security claims remain traceable to concrete enforcement surfaces. Alcyone, the Rust-based flight software architecture developed from those requirements, makes security boundaries explicit through service isolation, bounded authority, authenticated messaging, memory-safe implementation, and autonomous recovery mechanisms. Selective formal verification checks critical implementation invariants rather than relying on testing alone.

A software-in-the-loop evaluation drives adversarial scenarios and measures time to detect, time to recover, blast radius, recovery completeness, and functionality retained under attack. Across 13 paired comparative scenarios, Alcyone produced a qualifying onboard detection in 12 scenarios versus 3 for the cFS baseline. Mean detection time was 0.83 s for Alcyone versus 7.80 s for cFS over detected scenarios; mean resilience ratio was 0.941 versus 0.722; and mean blast radius was 1.3 versus 3.0 mission functions. Additional experiments characterize cross-level functionality retention, attack-intensity thresholds, phase sensitivity, and detector-configuration tradeoffs.

Cyber resilience in spacecraft flight software is an engineering problem, not a late-stage

compliance concern or an extension of conventional reliability practice. The resulting methodology links attack-surface analysis, requirement derivation, architectural design, and quantitative evaluation into a traceable basis for flight software that detects compromise, bounds propagation, and preserves mission function under adversarial stress.

Readers and Advisors: James G. Bellingham, Primary Advisor; Michael G. Ryschkewitsch, Co-Advisor; Patrick J. Binning, original co-advisor; Gregory J. Falco, original primary advisor; Anton T. Dahbura, committee chair.

Acknowledgments

I owe a special debt to Gregory J. Falco, who first encouraged me to apply to the Doctor of Engineering program at Johns Hopkins as his student and who has been my principal collaborator throughout this work. Much of this dissertation grew from the research agenda, technical questions, and publication path we developed together. His confidence in the project made it possible for me to begin it, and his collaboration shaped its direction.

I am deeply grateful to James G. Bellingham for stepping in as my primary advisor after Greg moved from Johns Hopkins to Cornell. His willingness to guide the dissertation through its final stages gave the project continuity, structure, and a clear path to completion.

I thank my APL co-advisors, Michael G. Ryschkewitsch and Patrick J. Binning, for their perspective on space systems engineering and for helping keep the work grounded in mission practice.

Anton T. Dahbura's detailed questions, cybersecurity perspective, and guidance helped clarify the dissertation's broader engineering contribution.

I am also grateful to Katherine S. Watko, who has supervised and supported my professional development at APL over many years, including before and during this doctoral work. She encouraged my education efforts, helped me connect the research back to mission needs, and gave me the flexibility to sustain the work alongside my responsibilities at the Laboratory.

The Johns Hopkins University Applied Physics Laboratory supported my doctoral study. APL's investment in continuing education made this degree possible, and the Laboratory's mission environment shaped many of the practical questions that motivated the work.

Portions of this dissertation are based on previously published or accepted papers coauthored with Gregory Falco and others. Material from those publications has been revised, reorganized, and integrated here to support the dissertation's broader argument. I acknowledge my coauthors' contributions to the underlying papers; responsibility for the dissertation's synthesis and any remaining errors is mine.

I also thank my parents, Billy and Judy, and my brother, Michael, for their love, encouragement, and confidence in me over many years. Their support helped make this path possible long before the dissertation itself began.

Finally, I thank my wife, Emily, and my daughter, Sophia. Emily, thank you for your patience, love, and steady belief in me through the long evenings, weekends, deadlines, and travel that this work required. I could not have finished this without you. Sophia, you gave me joy and perspective through the hardest parts of the process. This dissertation belongs to both of you as much as it belongs to me.

For Emily and Sophia, with all my love.

Table of Contents

Abstract . . . . **ii**

Acknowledgments . . . . **iv**

Dedication . . . . **v**

Table of Contents . . . . **vi**

List of Tables . . . . **xi**

List of Figures . . . . **xii**

1 Introduction **1**
- 1.1 Motivation and Thesis 1
- 1.2 Problem Statement and Research Gap 3
  - 1.2.1 Reliability Is Not Cyber Resilience 3
  - 1.2.2 Current Practice Leaves Key Gaps 4
  - 1.2.3 Future Missions Widen Those Gaps 6
- 1.3 Research Questions and Contributions 6
  - 1.3.1 Research Questions 6
  - 1.3.2 Summary of Contributions 7
- 1.4 Scope and Limitations 9
- 1.5 Organization of the Dissertation 10

2 Background **11**
- 2.1 Space Systems and Flight Software 11
  - 2.1.1 Historical Evolution of Spacecraft Computing 11
  - 2.1.2 Modern FSW Architectures and Mission Profiles 12
  - 2.1.3 Software Engineering Practices in Space 13
- 2.2 The Space Cybersecurity Landscape 14
  - 2.2.1 The Vacuum of Space Cybersecurity 14
  - 2.2.2 Emerging Policy and Frameworks 15
  - 2.2.3 Key Incidents and Threat Evolution 16
  - 2.2.4 Space-Specific Threat Frameworks 17
- 2.3 Cyber Resilience as an Engineering Lens 18
- 2.4 Existing FSW Frameworks 20
  - 2.4.1 NASA core Flight System (cFS) 20

2.4.2 JPL F' . . . 21
2.4.3 Microkernel and Separation Kernel Approaches . . . 22
2.5 Secure Coding and Invariant-Based Development . . . 23
2.5.1 The Memory Safety Imperative . . . 25
2.5.2 Rust for Aerospace . . . 26
2.6 Summary . . . 27

3 Threat Analysis and Attack Surface . . . 28
3.1 Related Work . . . 29
3.1.1 Complexity in Spacecraft Computing Systems . . . 29
3.1.2 Mapping a Spacecraft's Attack Surface . . . 30
3.1.3 SPARTA as the Space Threat Taxonomy . . . 32
3.2 Methodology: Bottom-Up Attack Surface Analysis . . . 32
3.2.1 Why a Bottom-Up Lens Matters . . . 32
3.2.2 Selection of Representative System . . . 33
3.2.3 Analysis Framework . . . 34
3.3 Attack Surface Findings in cFS on RTEMS . . . 36
3.3.1 OSAL Complexity and Opacity . . . 36
3.3.2 RTEMS Execution Model and Post-Exploitation Primitives . . . 37
3.3.3 BSP Configuration and Permissive Defaults . . . 38
3.3.4 Memory Safety as an Attack-Surface Multiplier . . . 39
3.4 From Surfaces to Techniques: Threat Mapping . . . 39
3.4.1 Threat Actors and Assumptions . . . 40
3.4.2 Mapping Representative Techniques to Flight-Software Surfaces . . . 40
3.4.3 Attack Chain Archetypes . . . 42
3.5 Summary . . . 44

4 Cyber Resilience Requirements Derivation . . . 46
4.1 Related Work . . . 47
4.1.1 Security Requirements as an Engineering Artifact . . . 47
4.1.2 Space Cybersecurity Standards and the FSW Requirements Gap . . . 48
4.1.3 The Secure-by-Component Standard for Space Systems . . . 49
4.1.4 Threat-Informed Requirements Approaches . . . 51
4.2 Secure-by-Component Methodology . . . 51
4.2.1 Method Overview . . . 51
4.2.2 Framing the Requirements Methodology . . . 53
4.2.3 Integrating the HAT TRICK Framework . . . 54
4.3 Applying Secure-by-Component to Command and Data Handling . . . 55
4.3.1 Decomposition of FSW Functions and Components . . . 56
4.3.2 Attack Surface Analysis . . . 56
4.3.3 Threat Analysis Using SPARTA and HAT TRICK . . . 58
4.3.4 Identification of Secure-by-Design Principles . . . 58
4.4 Testable Cyber Requirements Generation . . . 59
4.4.1 Redesign of Components into Secure Blocks . . . 59
4.4.2 Representative Requirements for Command Reception and Validation . . . 60

4.4.3 Per-Component Implementation Features . . . 62
4.4.4 Requirements Categories and Traceability . . . 63
4.4.5 Verification Approaches and Testability Criteria . . . 64
4.4.6 Tooling and Evolution . . . 65
4.5 From Requirements to Architecture Constraints . . . 66
4.6 Summary . . . 67

5 Alcyone: Cyber-Resilient FSW Architecture . . . 69
5.1 Related Work . . . 70
5.2 System Design Approach . . . 72
5.2.1 Core Design Process . . . 72
5.2.2 Mission Context and Scope . . . 73
5.2.3 Design Scope and Assumptions . . . 73
5.3 Driving Concept and Formal Requirements . . . 74
5.3.1 Cyber Resilience Principles . . . 74
5.3.2 Threat Model . . . 75
5.3.3 Requirements and Traceability Basis . . . 77
5.4 High-Level Architecture . . . 77
5.4.1 Command and Data Handling . . . 80
5.4.2 Supervision and Recovery . . . 84
5.4.3 Support Services and Mission/Application Layer . . . 85
5.4.4 Standards-Shaped External Interfaces . . . 87
5.4.5 Runtime and Hardware Abstraction . . . 88
5.5 Benefits of Rust for Flight Software . . . 89
5.5.1 Safety Guarantees . . . 89
5.5.2 Type System and Interface Contracts . . . 91
5.5.3 Performance, Portability, and Embedded Viability . . . 92
5.5.4 Adoption Constraints . . . 93
5.6 Verification and Validation Strategy . . . 94
5.6.1 Implementation-Level Assurance . . . 95
5.6.2 System-Level Verification in SWIL . . . 98
5.7 Comparison with cFS and F' Architectures . . . 99
5.8 Summary . . . 100

6 Experimental Evaluation . . . 102
6.1 Related Work . . . 103
6.1.1 Cyber Resilience Measurement and Metrics . . . 103
6.1.2 Adversarial Testing for Space Systems . . . 104
6.1.3 Fault Injection and Conventional FSW Testing . . . 105
6.2 Experimental Setup and Methodology . . . 106
6.2.1 Test Environment . . . 106
6.2.2 Attack Injection Framework . . . 107
6.2.3 Experiment Set . . . 108
6.2.4 Shared Scenario and Run Structure . . . 109
6.2.5 Common Analysis and Validity Rules . . . 111

6.3 Resilience Metrics . . . 112
6.4 Experiment 1: Comparative Architectural Resilience . . . 115
6.4.1 Design . . . 115
6.4.2 Results . . . 117
6.5 Experiment 2: Quantitative Measurement of Cyber-Resilience . . . 121
6.5.1 Design . . . 121
6.5.2 Results . . . 122
6.6 Experiment 3: Attack Intensity Threshold Sweeps . . . 125
6.6.1 Design . . . 125
6.6.2 Results . . . 125
6.7 Experiment 4: Attack Timing by Mission Phase . . . 127
6.7.1 Design . . . 127
6.7.2 Results . . . 127
6.8 Experiment 5: IDS Configuration Sensitivity and Ablation . . . 129
6.8.1 Design . . . 129
6.8.2 Results . . . 130
6.9 Threats to Validity . . . 134
6.9.1 Internal Validity . . . 134
6.9.2 External Validity . . . 135
6.9.3 Construct Validity . . . 135
6.9.4 Experimental Design Trade-offs . . . 136
6.10 Summary . . . 137

7 Discussion . . . 138
7.1 Answers to the Research Questions . . . 138
7.1.1 RQ1: Threat Landscape and Salient Attack Surfaces . . . 138
7.1.2 RQ2: Minimum and Testable Requirements . . . 139
7.1.3 RQ3: Architecture That Embeds Resilience by Construction . . . 140
7.1.4 RQ4: Evaluation and Measurement . . . 141
7.2 Technical Implications for Flight Software Architecture . . . 142
7.2.1 Enforcement Surfaces and Interface Policy . . . 142
7.2.2 Evidence Integrity and Measurement Credibility . . . 143
7.2.3 Implementation Assurance and Residual Engineering Limits . . . 144
7.3 Operational, Program, and Policy Implications . . . 145
7.3.1 Budgeted Autonomous Response . . . 145
7.3.2 Reuse, Requirements, and Risk as Program Gates . . . 146
7.3.3 Acquisition, Standards, and Lifecycle Governance . . . 146
7.4 Boundary of the Dissertation Claim . . . 147
7.4.1 What the Evidence Establishes . . . 147
7.4.2 What the Evidence Does Not Establish . . . 148
7.4.3 Generalization Boundary . . . 148
7.5 Future Work . . . 149
7.5.1 Space-Specific RTOS Evaluation and Replacement . . . 149
7.5.2 Hardware-Rooted Trust and Multi-Processor Architectures . . . 149
7.5.3 Formal Verification of Flight Software Components . . . 150

7.5.4 On-Board Anomaly Detection and Autonomous Response 150
7.5.5 Resource-Constrained Security 151
7.5.6 Compromised-Node Resilience for Distributed Space Systems 152
7.5.7 Testbed Extensions for Multi-Spacecraft and Hardware-in-the-Loop Validation 152
7.5.8 Cislunar and Proliferated Mission Class Extensions 153
7.5.9 Multi-Stakeholder Trust and Governance 153

8 Conclusion 154
8.1 Summary of Contributions 155
8.1.1 Research Framing and Problem Scope 155
8.1.2 Threat Landscape and Attack Surfaces 155
8.1.3 Minimum and Testable Requirements 156
8.1.4 Architecture by Construction 156
8.1.5 Evaluation and Measurement 157
8.2 Final Implications 158

References 159

A Acronyms 175

B Top-Level Cyber Resilience Requirements 178

C Curriculum Vitae 182

List of Tables

Table 2.1 Security-relevant properties of representative FSW frameworks 24

Table 4.1 Requirement category traceability 64
Table 4.2 Sextant artifact types managed for the Alcyone systems-engineering lifecycle. 66

Table 5.1 Alcyone cyber resilience principles 76
Table 5.2 Requirements-to-architecture traceability 78

Table 6.1 Comparative scenario suite 110
Table 6.2 Metric-to-NIST goal mapping 115
Table 6.3 Architectural comparison of the cFS and Alcyone baselines 116
Table 6.4 Scorecard results for Experiment 1 118
Table 6.5 Recovery results for Experiment 1 119
Table 6.6 Multi-level functionality result for Experiment 2 124
Table 6.7 Attack-intensity sweep results 126
Table 6.8 Phase-timing results 128
Table 6.9 Window-duration sweep results 130
Table 6.10 Threshold sweep results 132
Table 6.11 Detector ablation results 133

List of Figures

Figure 2.1 Typical cFS layered architecture 21

Figure 4.1 Secure-by-component assembly process 50
Figure 4.2 Example security fault tree anchored to a mission-level failure mode . . 52
Figure 4.3 C&DH-related data flows. 57

Figure 5.1 Alcyone layered architecture 79

Figure 6.1 Experiment 2 functionality-over-time traces by scenario 123
Figure 6.2 Experiment 2 ratio and attenuation summaries 124

Chapter 1
Introduction

1.1 Motivation and Thesis

Flight software is the on-board software stack that manages command and data handling, fault management, and mission control logic on a spacecraft's primary computer. Its reliability has been studied for decades through the lens of quality assurance and fault tolerance. Despite sustained attention to fault management, defending that same software against an intelligent adversary—rather than against probabilistic faults—has received far less.

Space is an increasingly important domain for national security, economic activity, scientific discovery, and national prestige; governments now recognize the need to protect critical space systems, with the United States designating space as critical national infrastructure. Central to that protective effort is the need for increased cyber resilience and cybersecurity in space systems, both in hardware and software [1].

Historically, spacecraft have not been a prominent target of cyber attack, so their developers have not built them with cyber resilience in mind. Falco [2] describes a "vacuum" of space cybersecurity in which systems central to critical infrastructure have historically lacked space-specific standards, shared threat understanding, and supporting institutions for systematically improving cybersecurity posture. Bailey [3] similarly argues that space cybersecurity must be approached as a threat-informed, risk-managed discipline: adversaries can pursue effects ranging from temporary disruption to irreversible loss of control, while the space environment compresses defender reaction time and complicates attribution and recovery.

Concrete incidents demonstrate that the threat is not hypothetical. In 1998, a cyber intrusion at NASA Goddard may have caused the German–American Röntgensatellit (ROSAT) X-ray satellite to point its sensor at the sun, destroying the instrument [4]. In 2007–2008, attackers

compromised the Svalbard ground station network and achieved all steps required to command NASA's Landsat-7 and Terra Earth-observation satellites [5]. In 2018, an unauthorized device on the Jet Propulsion Laboratory network allowed attackers to pivot into mission systems that interface with deep-space probes [6]. And in 2022, a Russia-attributed cyberattack against ViaSat KA-SAT ground terminals disrupted satellite communications across Ukraine and parts of Europe on the eve of the invasion [7]. Together these span distinct attack vectors—ground-station compromise, internal network pivoting, and user-segment disruption—and establish that consequences for operational missions range from data loss to permanent instrument destruction. The incidents share a pattern: the attacker exploited a surface that the designers did not explicitly defend. Reducing that class of outcomes is what cyber resilience is intended to address.

Space system developers have not yet had a domain-specific method for translating the National Institute of Standards and Technology's (NIST) guidance for developing cyber-resilient systems into flight-software requirements, architectures, and evidence [8]. That guidance is general by design. The burden falls on developers to tailor it to the embedded, real-time, resource-limited spacecraft environment and to express those methods in an engineering arc of anticipate, withstand, recover, and adapt.

Bailey [3] addresses part of that gap, distilling four cyber resilience principles applicable to spacecraft: being robust, being opaque, constraining behavior, and being responsive. Of these, robustness and constraint apply most directly to the software domain: flight software must resist attack, contain the propagation of a compromise that succeeds, and mitigate its effects. It must also constrain unsafe behaviors, so an attacker cannot co-opt components to produce harmful outcomes. Earlier work identifies flight software as a high-leverage point for resilience because compromise can affect spacecraft behavior directly: command handling, operating modes, telemetry, fault response, and payload operation [9].

Thesis. Cyber resilience can be engineered as a first-class system property of spacecraft flight software. Threat analysis can be translated into requirements, requirements into architec-

tural enforcement points, and enforcement points into measurable resilience outcomes. The chapters that follow decompose representative flight software attack surfaces and adversary scenarios, then derive minimum and testable cyber requirements tied to explicit enforcement points. From those requirements, they design and implement a secure-by-construction flight software architecture that contains compromise through isolation, memory-safe implementation, and verifiable enforcement mechanisms. They finally assess resilience outcomes under common adversarial conditions so the resulting architectural tradeoffs can be compared and refined.

1.2 Problem Statement and Research Gap

Space system developers have only limited, unevenly adopted methods for building cyber resilience into spacecraft flight software or into the mission systems that depend on it. Reliability engineering, the discipline that has historically dominated flight software, does not by itself yield cyber resilience. Prevailing development practice leaves key resilience objectives unaddressed at the architectural level, and the operational environments that future missions inhabit make ground-dependent recovery less and less viable [3].

1.2.1 Reliability Is Not Cyber Resilience

Flight software has historically been engineered for correctness and fail-safe operation, with emphasis on redundancy, verification, and bounded resource use. Those practices are essential for mission assurance, but they are not sufficient for cyber resilience: they assume failures arise from probabilistic environmental effects or component faults rather than from an intelligent adversary deliberately selecting inputs and execution paths to defeat defenses. A system can be robust against random upsets and remain brittle under adversarial manipulation.

Fault management inherits a set of assumptions from reliability engineering: that failures fall into predictable classes, arrive in predictable sequences (radiation upsets, component

wear-out), and can be recovered from by preplanned procedures. Cyber threats violate each of those assumptions. An adversary chooses inputs specifically because they fall outside the modeled fault set, observes how the system responds, and adapts. A fault-management system designed to recognize stuck-bit patterns will not recognize a crafted command sequence that produces the same telemetry signature while accomplishing something else. Resilience therefore requires containment boundaries and recovery behaviors that remain meaningful when the failure process is being steered, not passively observed.

Flight software is a high-value target because access to it can yield partial or total control of spacecraft functions. Because spacecraft are accessed remotely, a secure ground system does not guarantee a secure spacecraft. Cyber attacks can also be attractive because they may be comparatively inexpensive and difficult to attribute when effects propagate through ground or supply-chain infrastructure.

1.2.2 Current Practice Leaves Key Gaps

Most flight software has been developed with reliability, determinism, and mission functionality as primary objectives, while cybersecurity has been treated as secondary or external to the architecture itself. That prioritization was easier to sustain when mission software stacks were bespoke and difficult to study. It is no longer sustainable in an environment of shared frameworks, reusable commercial off-the-shelf platforms, open-source codebases, and converging toolchains: National Aeronautics and Space Administration (NASA)'s core Flight System (cFS) and Jet Propulsion Laboratory's F' are both open-source flight software stacks available to defenders and adversaries alike, which means security properties must be engineered and evidenced rather than assumed. The security posture of a spacecraft cannot depend on keeping its code and interfaces secret. Flight software must be designed so that compromise of a single interface or component does not imply loss of the mission, and so that degraded but safe operation remains possible when prevention fails. That constraint changes the architectural question from whether compromise can be prevented absolutely to whether it can be bounded.

Building cyber resilience therefore requires more than generic secure-coding guidance; it requires understanding the threat landscape, the adversary organizations that may target a system, and the tactics and techniques available to them. Open-source literature on cyber attacks against spacecraft remains sparse. The Space Attack Research and Tactic Analysis (SPARTA) project was developed to document and categorize threats against spacecraft, but it is still maturing and requires ongoing validation and extension. The thinness of the incident record is one obstacle; the deeper one is the absence of a mature, flight-software-specific method for translating architecture, interfaces, and operational dependencies into a defensible attack-surface model. That gap motivates the research agenda in earlier work [9] and is addressed in detail in Chapter 3.

Organizational and lifecycle gaps compound these architectural ones. Developers of flight software—civilian agencies, universities, defense contractors, and commercial providers—rarely specialize in cybersecurity, and their development lifecycles emphasize safety, mission assurance, and delivery under severe resource constraints rather than explicit adversarial modeling. Conventional verification and validation show that software behaves correctly under expected conditions, but they do not establish that the same software can detect malicious inputs, contain a compromised component, or recover meaningfully after adversarial manipulation. As a result, cyber resilience needs are discovered late, addressed unevenly across subsystems, or captured as mission-specific add-ons rather than as architectural requirements from the outset.

The sharpest gaps appear at the implementation layer. Flight software runs in tightly constrained computing, networking, storage, power, and thermal envelopes, and the attestation, monitoring, and response capabilities that enterprise security controls assume are largely absent—with no drop-in replacements engineered for the embedded mission context. These practice gaps are compounded by the dominant implementation languages: most existing flight software is written in C and C++, which leave memory safety to developer discipline even as government and research organizations increasingly identify memory unsafety as a class of vulnerability to eliminate [10, 11].

1.2.3 Future Missions Widen Those Gaps

The challenge of securing flight software is compounded by the expansion of human activity beyond traditional orbits, where existing operational assumptions about response time and connectivity no longer hold. Prior cislunar analysis argues that long communication delays, intermittent contact windows, multi-hop networking, and more complicated trust relationships shift responsibility for detection, containment, and recovery onto on-board systems themselves [12]. Bailey makes a compatible point from a different direction: because defender reaction time is compressed and adversary effects can escalate rapidly, space cybersecurity must be approached as a threat-informed, risk-managed discipline [3].

The significance of the broader context is that future mission classes make existing flight-software security gaps less tolerable. Architectures that rely on implicit trust or continuous ground intervention become less viable as autonomy and operational distance increase. Cyber resilience must therefore be engineered as an on-board property of flight software rather than treated as an external operational safeguard.

1.3 Research Questions and Contributions

Flight software security depends on connecting threat analysis, requirements, architecture, and evidence — an engineering problem, not a compliance one. Four research questions organize that argument and define the contributions that follow.

1.3.1 Research Questions

Four research questions organize the work:

RQ1: Threat landscape. What attack surfaces and threat scenarios are most consequential for mission integrity in contemporary, widely used flight software stacks, and what do those surfaces imply about realistic adversary actions?

RQ2: Minimum requirements. What minimum set of secure-by-design principles and requirements are necessary to preserve mission-critical control and availability, and how can those principles be translated into requirements that are meaningfully testable?

RQ3: Architecture. How can an architecture embed resilience by construction, using explicit trust boundaries, isolation, and enforcement surfaces so that the system can contain compromise and support autonomous recovery?

RQ4: Measurement. How can cyber resilience in flight software be measured quantitatively in a way that enables comparison across architectures and provides evidence for or against specific design choices?

1.3.2 Summary of Contributions

The contributions form an end-to-end engineering program for cyber-resilient flight software. The program begins by sharpening the problem space: a research agenda for flight software security [9] and a memory-safety analysis that treats implementation language choice as a resilience question rather than only a productivity or software-quality concern.

It then contributes an attack-surface analysis of a representative open-source stack (cFS on Real-Time Executive for Multiprocessor Systems (RTEMS)) that identifies inherited and emergent vulnerability classes and maps them to mitigation leverage points. It establishes a concrete method for performing attack-surface analysis on flight software as such, rather than borrowing unmodified techniques from other software domains [13].

A two-stage requirements contribution follows: a minimum-requirements methodology derives a mission-preserving set of secure-by-design principles from permanent loss-of-control scenarios, and a secure-by-component workflow translates those principles into traceable, testable requirements for flight-software subsystems [14, 15].

Alcyone realizes those requirements in a flight software system, delivered as both an architecture and a working software artifact. Alcyone embeds cyber resilience in enforcement

boundaries such as isolation, zero-trust messaging, bounded recovery, and memory-safe implementation, while formal verification checks critical invariants rather than relying exclusively on testing [16].

It also contributes experimental infrastructure for evaluating these claims: an adversarial software-in-the-loop testbed, named *Taurus*, and a structured comparative benchmark that evaluates architectures under common threat scenarios. The evaluation compares architectural paradigms in a series of campaigns that pit Alcyone's design choices—process isolation, per-command authorization, compile-time memory safety—against current operational flight software.

Three contributions are novel in their own right. The first is the bottom-up attack-surface methodology, which begins from real-time operating system (RTOS) and abstraction-layer surfaces rather than adversary techniques (Chapter 3). The second is Taurus, the software-in-the-loop (SWIL) adversarial testbed, which drives SPARTA-mapped scenarios through the Consultative Committee for Space Data Systems (CCSDS) command boundary with no agent on the flight target (Chapter 6). The third is the multi-level functionality-based resilience measurement adapted from Weisman et al. for spacecraft (Chapter 6). The remaining contributions derive their force primarily from the chain composition rather than from individual novelty.

The published-paper portion of the research program has produced six peer-reviewed conference papers, each backing one of these contributions:

- *A Research Agenda for Space Flight Software Security* [9] — IEEE SMC-IT 2023; motivates the program and defines the resilience framing (§ 1.1).
- *Attack Surface Analysis for Spacecraft Flight Software* [13] — IEEE SMC-IT 2024; supports Chapter 3.
- *Minimum Requirements for Space System Cybersecurity — Ensuring Cyber Access to Space* [14] — IEEE SMC-IT 2024; supports Chapter 4.
- *Testable Cyber Requirements for Space Flight Software* [15] — IEEE Aerospace Conference

2025; supports Chapter 4.

- *Alcyone: A Blueprint for Secure Rust Flight Software* [16] — IEEE SMC-IT 2025; supports Chapter 5.
- *Cyber Resilience in Cislunar Space: Security Strategies for Large-Scale Space Infrastructure* [12] — IEEE SMC-IT 2025; broader-applicability extension.

1.4 Scope and Limitations

Flight software — defined in Section 1.1 — is the primary object of analysis. Chapter 2 decomposes its layers and interfaces. Other spacecraft-related software (embedded firmware, instrument and payload software, ground-segment systems, and link-layer cryptography) is important but falls outside the primary scope, referenced only to clarify system boundaries, threat assumptions, and the role of flight software within a broader cyber-physical mission. The four research questions of Section 1.3 provide a single thread from threat analysis through requirements and implementation to experimental assessment.

Several internal limitations bound the scope of the dissertation. First, the attack-surface analysis targets a representative baseline, cFS on RTEMS, deployed across a significant class of civil and scientific spacecraft but not generalizable to all flight computing environments; the proposed Alcyone architecture is a separate, independently scoped contribution. Proprietary real-time operating systems, multi-core heterogeneous processors, and deep-space relay architectures may present different attack surfaces and recovery constraints. Second, the threat model is explicitly bounded: it emphasizes software-exploitable vulnerabilities accessible through the command uplink and software-defined mission interfaces, and does not treat hardware fault injection, supply-chain compromise, or cryptographic attacks as primary vectors. Third, the empirical assessment is scoped to a specific adversarial scenario set implemented in the Taurus testbed; conclusions about which design choices improve resilience are valid within that scenario envelope and should not be extrapolated to hardware fault-injection, cryptographic

compromise, or supply-chain threat classes without separate evaluations designed for each.

The cislunar analysis is bounded similarly. It is used as a broader-applicability extension that examines how the flight-software arguments change when operational latency, network indirection, and multi-stakeholder governance become dominant constraints. It is not intended as a second primary empirical subject on equal footing with the flight-software attack-surface, requirements, architecture, and evaluation chapters.

1.5 Organization of the Dissertation

The dissertation follows a resilience evidence chain. Chapter 2 defines flight software, cyber resilience, and the implementation constraints that shape the rest of the argument. The middle chapters then move from adversary leverage to enforceable design: Chapter 3 analyzes attack surface in a representative open-source stack, Chapter 4 turns those findings into minimum and testable requirements, and Chapter 5 realizes them in Alcyone's architecture. Chapter 6 tests whether those architectural choices change measured behavior under adversarial stress. Chapter 7 draws the implications for space cybersecurity practice and standardization, and Chapter 8 closes by synthesizing the contributions and remaining limits.

Chapter 2
Background

Chapter Abstract. Flight software sits within a broader space mission stack whose security properties depend on on-board control layers, framework boundaries, memory-safety choices, and implementation language. Flight-software layers and mission profiles, cyber resilience as the engineering frame, representative open-source frameworks, and Rust's role in safety-critical aerospace are the four framing pieces this chapter develops.

This framing draws on earlier work [9] that identified open challenges in flight software security and proposed research directions developed across the technical chapters. Each technical chapter (Chapters 3–6) develops its own focused related work.

2.1 Space Systems and Flight Software

Space missions embed software across many layers of the vehicle and supporting infrastructure. The on-board control environment is the central focus because compromised flight software can alter command handling, operating modes, telemetry, fault response, and payload operation.

2.1.1 Historical Evolution of Spacecraft Computing

Flight software has evolved from tightly scoped control programs into systems that resemble on-board operating environments, but the expectation of deterministic behavior under harsh conditions has not changed. What has grown is scale: a modern Mars landing mission combined several million lines of flight code with substantial heritage reuse, executing on a RAD750-class processor whose memory and clock speed are modest by modern server standards [17]. Security mechanisms that assume frequent updates, high-bandwidth telemetry, or heavy runtime monitoring must adapt to these constraints; complexity also extends beyond lines of code into

toolchains, generated code, and dependency structure, making compilers, build scripts, and third-party components part of the assurance boundary.

2.1.2 Modern FSW Architectures and Mission Profiles

Flight software is a layered control stack: an operating system (traditionally a real-time operating system, but increasingly a commodity system such as Linux) and platform support code below, middleware and framework services in the middle, and mission applications above [13]. An *attack surface* is the set of points at which an attacker can attempt to access, influence, or extract data from a system [18]; for flight software, this extends well beyond the radio uplink to include internal buses, file systems, and platform configuration at every layer of that stack. A *trust boundary* is any interface across which one component relies on the correctness or integrity of another without independent verification; crossing one without validation is the mechanism by which a localized compromise becomes system-wide. Attack surface is often inherited upward through dependencies, and trust boundaries are frequently implicit; later chapters argue for making them explicit and enforceable.

Mission profile shapes both engineering constraints and defensive assumptions. In low Earth orbit (LEO), regular contact lets operators diagnose anomalies, downlink detailed telemetry, and push configuration updates. In deep-space and cislunar regimes, communication delays and intermittent links shift the burden to on-board autonomy: detection, containment, and recovery must occur without immediate ground intervention [8, 19]. Resilience mechanisms under those conditions are not optional hardening; they are part of the control architecture.

Because flight software translates commands into actuator outputs and mission-critical behaviors, access to it yields partial or total control of spacecraft functions, which adversaries can then co-opt for their own goals. Since authorized users access flight software remotely, a hardened ground system does not mean a spacecraft is secure: any path that reaches the vehicle's uplink interface or software update mechanism becomes a potential attack vector.

2.1.3 Software Engineering Practices in Space

Heritage reuse is not simply cultural inertia; it is a rational response to verification cost and mission risk. Major programs frequently combine newly written code with existing components that have accumulated evidence through prior test campaigns and flight history [17]. This practice reduces schedule and safety risk, but it can also propagate vulnerabilities and insecure assumptions across generations. Cyber resilience therefore cannot depend on "starting over"; it must be achievable through systematic analysis, requirements, and architectural controls that apply to both new and inherited components.

Space vehicle development projects, including flight software, typically carry associated sets of detailed requirements and engineering guidelines. These standards require that the software meets the needs of mission owners, implements the required technical functions, and executes according to its intended design with risk that is identified and mitigated. Examples include NASA's Software Engineering Requirements [20], the European Space Agency (ESA) code standards [21], and JPL's standards for using the C programming language [22].

Space software verification and validation (V&V) is typically disciplined and evidence-driven, with standards emphasizing requirements traceability, configuration control, and independent verification activities. In high-consequence settings, teams employ extensive static analysis and review: one representative mission integrated multiple static analyzers and custom checkers into routine builds, and reviewers processed substantial volumes of peer comments and tool findings [17].

Existing quality standards focus on verifying that software meets its intended design and do not address resilience against cyber attacks. Cybersecurity requirements, when they exist for flight software systems, are often highly technical, focused on compliance, and lack traceability to mission needs or specific cyber threats [23]. Because cyber threats evolve faster than complex systems, these systems can be insecure on day one of operations. Existing standards remain necessary as a foundation but are insufficient for addressing cyber resilience.

Flight software operates under constraints that reduce the feasibility of post-deployment

remediation. Real-time deadlines, limited power and compute, long mission lifetimes, and the absence of physical access constrain patch cadence and forensic visibility [17]. Under these conditions, it is more defensible to prevent dominant vulnerability classes by construction and to design explicit containment and recovery behaviors than to rely on post-compromise remediation.

2.2 The Space Cybersecurity Landscape

The cybersecurity context for flight software has evolved on both technical and institutional fronts. Although the analysis centers on the on-board stack, the threat model spans the mission system: development toolchains, ground interfaces, and operational workflows shape what adversaries can reach and how defenders can respond.

2.2.1 The Vacuum of Space Cybersecurity

Falco [2] characterizes a "vacuum" of space cybersecurity: security considerations were historically peripheral to space systems engineering, and the sector lagged other critical infrastructure domains in standards, practices, and shared threat understanding. The vacuum is partly institutional (limited public disclosure, fragmented ownership, and safety-dominant assurance culture) and partly technical (mission-unique stacks that resist standardization). Few cybersecurity standards address the space domain specifically, and flight software organizations rarely maintain deep cybersecurity expertise [2, 24]. The legacy effect remains visible in flight software architectures and development processes that were not designed with explicit adversary models in mind. Thummala et al. [25] formalize this distinctiveness as seven mission-level constraints—heterogeneity, physics coupling, immutability, mandatory autonomy, environmental degradation, subsystem interdependence, and governance pressure—arguing that while each constraint appears in other domains, their simultaneous presence in space missions produces a cyber resilience problem that cannot be addressed by adapting terrestrial frameworks wholesale.

Flight software development spans civilian agencies, universities, defense contractors, and commercial providers, each with different risk profiles, development cultures, and cyber resilience needs. Traditional assumptions — physical and logical isolation, trusted supply chains, and the belief that specialized protocols and mission uniqueness provided a degree of protection [2] — are increasingly brittle. Shared flight software frameworks, converging embedded toolchains, and spacecraft operations integrated with terrestrial networks all reduce the plausibility of obscurity as a defense [19]. Flight software must therefore be engineered as if adversaries can study its structure and as if compromise pathways can originate outside the vehicle.

2.2.2 Emerging Policy and Frameworks

Space Policy Directive-5 (SPD-5) formalized U.S. cybersecurity principles for space systems and signaled that cybersecurity is a policy concern, not merely an implementation detail [26]. Subsequent executive orders and operator guidance extended that baseline into acquisition, civil-space operations, and critical-infrastructure practice [27, 28, 29].

International activity has accelerated in parallel. The European Union Agency for Cybersecurity (ENISA) has published a Space Threat Landscape report mapping threat actors, attack techniques, and control frameworks to space-system assets [30]. Germany's Federal Office for Information Security (BSI) has released TR-03184 on information security for space systems [31], and Japan's Ministry of Economy, Trade and Industry (METI) has issued cybersecurity guidelines for commercial space systems [32]. Space ISAC partnerships and regional hubs show the same trend toward shared operational coordination [33, 34]. For flight software, the durable point is not the status of any one initiative but the convergence itself: security-relevant requirements and assurance evidence are increasingly treated as lifecycle and acquisition artifacts rather than as late implementation add-ons.

Recent U.S. policy guidance has explicitly called for reducing the attack surface associated with memory-unsafe languages, including by adopting memory-safe languages where feasible

[11]. In space missions, this intersects with long lifetimes and limited patch opportunities: eliminating dominant classes of vulnerabilities through language-level guarantees can be an enabling condition for resilience rather than an optional optimization.

NIST cybersecurity and cyber resilience guidance is not space-specific, but it provides the general engineering vocabulary used here for trustworthiness, resiliency goals, and implementation techniques [8]. NISTIR 8270 treats satellite operations, NISTIR 8401 addresses ground-segment command and control, and NISTIR 8441 extends the same concern to hybrid satellite networks [35, 36, 37]. Complementary best-practice guides from NASA adapt this framing to civil space operators [38].

National-security and practitioner guidance tailors the same control landscape more directly to space platforms. CNSSP 12 and the CNSSI 1253 Space Platform Overlay define policy and overlay expectations for national security space systems [39, 40], while the Aerospace Corporation's Space Segment Cybersecurity Profile provides a practitioner-facing tailoring of the same control catalogs [41]. A recurring challenge remains: turning broad control catalogs and resilience techniques into flight-software-relevant requirements that are testable and that map to enforceable architectural decisions.

IEEE P3349 is one coordinated standardization effort aimed at translating this policy and control landscape into a technical standard for space system cybersecurity [42].[1] The dissertation's contributions are therefore positioned alongside emerging standardization work rather than as a replacement for it.

2.2.3 Key Incidents and Threat Evolution

The public record of cyber incidents against space systems spans more than two decades and shows a progression from opportunistic ground-segment intrusions to sophisticated, targeted operations. The 1998 ROSAT incident is widely cited as an early case study in which ground-segment compromise propagated to physical spacecraft damage [4]—exactly the failure mode

[1] The author participated in the IEEE P3349 working group and served on its steering committee. The minimum-requirements methodology developed in Chapter 4 fed into the standard's requirements prioritization.

that motivates on-board resilience mechanisms independent of ground-segment integrity.

At CYSAT 2023, a Thales research team seized control of ESA's OPS-SAT demonstration satellite in a sanctioned exercise, the first publicly disclosed ethical hack of an in-orbit spacecraft [43, 44]. Zhang et al. [45] subsequently demonstrated passive eavesdropping on geostationary satellite links at scale, recovering sensitive internal network traffic from unencrypted SATCOM downlinks. Attribution is often impossible from open sources, which argues for conservative threat assumptions and architectures that remain safe under uncertainty. The stakes are high: space systems provide leverage over communications, navigation, intelligence, and economic activity [3], and commercialization continues to expand the set of stakeholders and supply-chain paths that can introduce vulnerabilities.

2.2.4 Space-Specific Threat Frameworks

Bailey documents NASA's IV&V cyber range efforts as a shift toward space-specific cyber testing and evaluation, acknowledging that testbeds and adversarial exercises are needed to complement static assurance artifacts [46].

SPARTA provides a space-specific threat taxonomy intended to help practitioners reason about common tactics and techniques in space missions and infrastructure [47]. SPARTA supports traceability from identified surfaces to plausible threat actions in the threat-scenario analysis of Chapter 3, and exemplifies the broader threat-based approach: identifying protections as a function of plausible adversary actions rather than as a generic checklist [3].

In terrestrial operational technology, the MITRE Adversarial Tactics, Techniques, and Common Knowledge (ATT&CK) for industrial control systems (ICS) framework provides a knowledge base of adversary tactics and techniques observed in attacks against industrial control systems [48]. Although ICS environments share constraints with spacecraft (real-time operation, limited computational resources, and safety-critical functions), ATT&CK for ICS does not address space-specific factors such as CCSDS protocol stacks, autonomous operations during communication blackouts, or the infeasibility of physical access for incident response. SPARTA fills

this gap as the space-domain counterpart: a structured taxonomy of tactics, techniques, and countermeasures specific to space missions.

Complementing these threat taxonomies, the High Adversary Tier Threat Response Interdicting Cyberspace Kill-chain (HAT TRICK) framework provides a structured set of 53 cybersecurity requirements designed for national security systems [49]. Unlike SPARTA, which catalogs adversary behavior, HAT TRICK specifies defensive requirements intended to counter zero-day exploits and advanced persistent threats at the system level. HAT TRICK is not space-specific; it was developed at the Johns Hopkins University Applied Physics Laboratory (JHU/APL) as a general-purpose cybersecurity requirements reference guide for national security systems. Spacecraft share many of the same resilience demands (high consequence of failure, exposure to sophisticated adversaries, and limited opportunities for remediation), which makes the framework directly applicable. Chapter 4 adapts a subset of HAT TRICK requirements to flight software.

Even taken together, these frameworks do not fully bridge space-specific threat taxonomies with certifiable, testable software requirements at the level of flight software interfaces and enforcement points. SPARTA catalogs what adversaries do; HAT TRICK specifies what defenders must achieve; neither produces implementation-ready requirements tied to specific software interfaces. This gap motivates the requirements methodology of Chapter 4.

2.3 Cyber Resilience as an Engineering Lens

Cyber resilience recasts the engineering question from whether compromise can be prevented to what the system does when compromise occurs. For flight software, that distinction is operationally decisive: it determines whether a spacecraft can continue its mission, degrade safely, or fail catastrophically under adversarial pressure. Resilience engineering answers this as a first-class system property rather than as post-hoc mitigation [8]. A first-class resilience property is specified, designed, verified, and traded off through the same engineering disciplines

that produce evidence for safety and reliability; it is not treated as a separate compliance overlay. The cyber-physical systems literature has converged on a compatible framing: cyber resilience is the ability of a system to prepare for, absorb, recover from, and adapt to the adverse effects of cyber-threats while preserving its core mission functionality [50].

Trustworthy systems engineering encompasses a family of frameworks for developing systems that merit justified confidence in their behavior under both nominal and adverse conditions. In airborne software, DO-178C provides assurance through development process rigor and structural coverage analysis [51]. In information security, the Common Criteria (ISO/IEC 15408) defines evaluation assurance levels for security-critical products. In European space missions, ECSS-E-ST-40C mandates software development and verification practices scaled to mission criticality [21]. Each framework addresses a different facet of trustworthiness (functional correctness, security evaluation, or development rigor), but none addresses the challenge of sustaining mission-critical functions under active cyber attack.

NIST SP 800-160 is the most relevant reference here and is published in two complementary volumes. Volume 1 defines the systems security engineering framework: a lifecycle discipline for designing, developing, and operating systems with justified trust properties, anchored in explicit requirements and evidence [52]. Volume 2 extends that framework to the adversarial case, defining a cyber resiliency engineering framework that treats adversarial compromise as a design-basis condition rather than an exceptional event [8]. The NIST framing connects threat assumptions to architectural techniques and measurable outcomes; subsequent chapters draw on both volumes—Volume 1 for requirements engineering (Chapter 4) and Volume 2 for the resilience-technique catalog that structures the architecture (Chapter 5).

NIST frames cyber resiliency around four complementary (not sequential) goals: anticipate adverse conditions; withstand their effects; recover critical functions; and adapt over time to reduce future impact [8]. The accompanying technique catalog provides concrete levers for engineering each goal—segmentation and isolation, redundancy, non-persistence, substantiated integrity, and adaptive response, among others.

Recent work has begun to formalize space cyber risk analysis at the methodology level. Ear [53] develops a graph-theoretic mission model that captures cascading effects of cyber compromise across mission control flows and data flows, supplies algorithms for mission risk analysis and mission hardening, and validates them on a satellite communications testbed against three real-world attack patterns. That line of work is complementary: Ear [53] provides analytical machinery for assessing and assigning risk treatments, and later chapters develop the architectural mechanisms by which a flight software system can enforce the boundaries that such assessments would otherwise have to recommend.

2.4 Existing FSW Frameworks

cFS and F' serve as representative open-source flight software frameworks for the threat analysis and architectural discussion that follow. The goal is not to indict any particular framework, but to use widely deployed baselines to reason concretely about attack surface, trust boundaries, and design tradeoffs under realistic constraints. The underlying real-time operating system is treated as a property of the deployment rather than as a separate object of background analysis; Chapter 3 examines the security implications of the RTEMS configurations typical for cFS in detail.

2.4.1 NASA core Flight System (cFS)

cFS is a layered framework with a core flight executive (cFE) providing common services, an operating system abstraction layer (Operating System Abstraction Layer (OSAL)), a platform support package (PSP), and an application layer that hosts mission-specific components (Figure 2.1) [13]. It has been flown on numerous missions and is widely adopted across organizations in part because it is open-source and supports common spacecraft software needs. cFS commonly runs on RTEMS, a Portable Operating System Interface (POSIX)-compatible real-time operating system in which tasks share an address space without hardware-enforced

isolation [13]. cFS optimizes for portability and reuse across platforms. From a resilience perspective, that reuse also makes inherited dependencies and configuration choices, including the underlying RTOS, part of the mission attack surface rather than incidental implementation detail.

Like many flight software frameworks, cFS has emphasized reliability and mission robustness rather than explicit adversarial resilience. Chapter 3 examines how specific cFE, Software Bus, and fault-tolerance design choices manifest as attack surface in representative deployments.

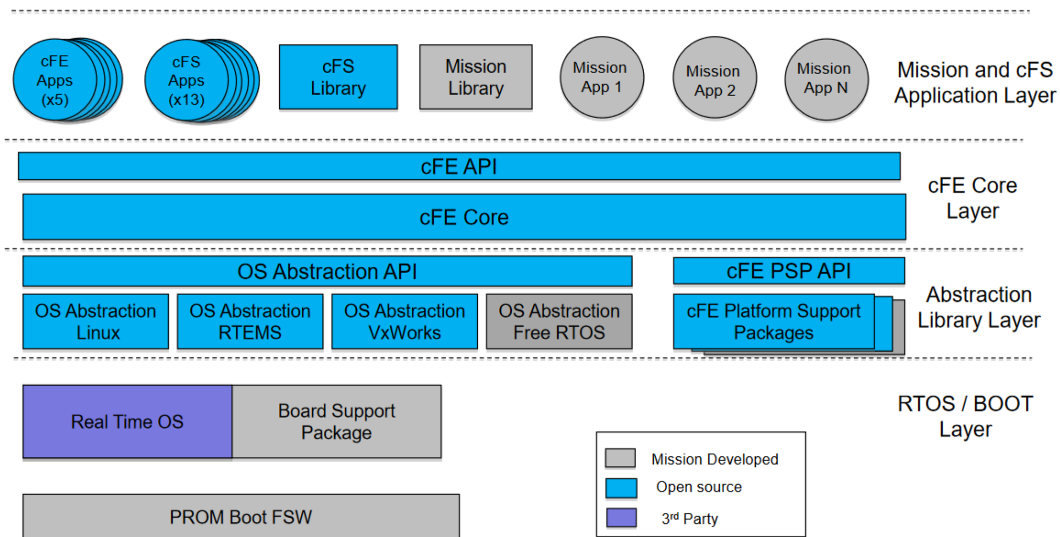


Figure 2.1 Typical cFS layered architecture showing mission applications, cFE core services, OS abstraction layer, and RTOS/BSP foundation. Open-source framework components (blue) provide common services; mission-developed components (gray) implement mission-specific logic. Adapted from Curbo and Falco [13].

2.4.2 JPL F'

F' provides a component-based model in which components communicate through typed ports and are composed into topologies, with tooling support for code generation and integration. Its use in the Mars Helicopter (Ingenuity) ecosystem demonstrates applicability to real missions. The structure improves modularity but produces no security boundaries; isolation and trust assumptions must be engineered deliberately.

The static topology model used by F' offers a security-relevant property that cFS's dynamic pub/sub does not: because port connections are defined at build time in the Framework Prime Processing (FPP) modeling language, all possible message flows between components are enumerable and can be analyzed before deployment. However, this analyzability does not translate to enforcement. Connected components communicate through typed ports without authentication or authorization checks, and the framework provides no mechanism to prevent a compromised component from invoking any port to which it has a static connection. The trust model is cooperative: components are assumed to behave correctly once connected.

Both cFS and F' host mission applications with their own security-relevant surface. Multiple applications run concurrently, communicating with each other, the RTOS, and spacecraft subsystems; any processing of unvalidated input, including serialization and deserialization of protocol traffic, is a potential attack vector. Shared middleware APIs and code libraries can propagate a single vulnerability across every subsystem that depends on them. An independent vulnerability assessment of cFS Aquila identified remote code execution, denial-of-service, and path traversal vulnerabilities in the framework's default configuration, confirming that inherited attack surface in open-source flight software stacks is not theoretical [54]. On spacecraft with multiple payloads or hosted instruments, the interfaces between separately developed firmware stacks deserve scrutiny where communication buses lack authentication.

2.4.3 Microkernel and Separation Kernel Approaches

The shared-address-space execution model used by cFS on RTEMS provides no hardware-enforced boundary between components. Microkernel and separation kernel architectures take a different approach by enforcing isolation at the operating system level.

The secure L4 microkernel (seL4) microkernel provides formally verified isolation guarantees: its capability-based access control model ensures that a component can only access resources for which it holds an explicit, unforgeable capability, and the kernel's functional correctness has been mechanically proved in Isabelle/HOL [55]. These properties make seL4 attractive as a

foundation for security-critical space systems, where the cost of a single compromise can be disproportionately high.

Jero et al. [56] describe the experience of porting NASA's cFS to seL4, finding that despite the OSAL's intended portability, significant OS-level functionality had to be created on top of seL4's minimal kernel interface. This effort led to the creation of Magnetite, an seL4-based operating system designed specifically for satellite software stacks. Magnetite implements OS services in Rust — a memory-safe systems programming language with ownership-based static memory management — on top of seL4's verified kernel, combining language-level memory safety with kernel-level isolation enforcement. The project demonstrates that building a secure satellite software stack requires attention not only to the flight software layer but also to the OS services beneath it.

Against that isolation context, the comparison narrows to two representative open-source flight software frameworks selected for analysis. The table has a limited purpose: it highlights which security-relevant properties are and are not enforced in cFS and F', not which kernel substrate is strongest in the abstract.

Table 2.1 summarizes the security-relevant properties of cFS and F'. Neither framework provides architectural enforcement of security boundaries at the messaging, command, or isolation level; both rely on cooperative trust between components and leave security enforcement to mission-specific implementation decisions.

2.5 Secure Coding and Invariant-Based Development

Programming languages, type systems, and development ecosystems can eliminate entire classes of vulnerabilities at the source rather than detecting and patching them after the fact. Kern formalizes this approach as *safe coding*: a collection of design patterns that shift safety responsibility from individual developers to programming languages, libraries, and frameworks by encapsulating risky operations within safe abstractions [57]. The key insight is that difficult

Table 2.1 Security-relevant properties of the two open-source flight software frameworks used as comparative baselines: cFS and F'.

Property	**cFS**	**F'**
Routing model	Dynamic pub/sub (runtime)	Static ports (build-time)
Message authentication	None	None
Command authority	Distributed (any app)	Distributed (any component)
Isolation model	Shared address space (OSAL)	Shared address space
Health authority	Distributed (H&S app)	Distributed (self-report)
Language / memory safety	C (unsafe)	C++ (unsafe)

safety reasoning can be localized to the implementation of safe abstractions, which undergo focused expert scrutiny, while the composition of these abstractions with the majority of the codebase is automatically verified by the language's type system. Applied at Google, this approach has nearly eliminated vulnerability classes including cross-site scripting and SQL injection [58].

The orientation is consistent with the NIST Secure Software Development Framework (SSDF), which prescribes process-level practices (defined security requirements, well-secured software production, vulnerability response) intended to be enforced through development tooling rather than left to per-developer discipline [59]. The same principle applies to memory safety: a memory-safe language encapsulates all memory operations within compiler-enforced abstractions that make violation structurally impossible.

Quantitative evidence supports this framing. Analysis of the Chromium browser codebase found that approximately 70% of serious security bugs are memory safety problems [60]. Google's experience with Android is sharper still: after progressively adopting Rust for new native code, the proportion of memory safety vulnerabilities in Android dropped from 76% in 2019 to 24% in 2024, with Rust code exhibiting less than half the emergency revert rate of C++ code [61]. Google's "Secure by Design" perspective explicitly argues that high-assurance memory safety can only be achieved through full adoption of languages with rigorous memory

safety guarantees [62]. For flight software, with constrained patch cadence and disproportionate exploit consequences, language-level safety is a first-order design decision rather than an implementation detail.

2.5.1 The Memory Safety Imperative

Memory-safe software does not reference memory outside its allocated space nor execute code outside its designated memory area. The two dominant error classes, allocation errors (use-after-free, double-free, uninitialized memory) and access errors (buffer overflows, out-of-bounds reads), have been a persistent source of security vulnerabilities in systems software. In spacecraft, where the margin for error is slim and patchability is constrained, a single memory corruption can affect mission behavior directly: incorrect trajectory calculations, disrupted command processing, or loss of telemetry integrity.

The C language has been a mainstay in space system development due to its broad feature set, heritage-based reuse culture, and longstanding presence in embedded engineering. C and C++ are not inherently memory-safe and leave the responsibility of managing memory access and allocation to programmers, a practice that leads to insecure patterns when individual developers devise their own protections. Secure coding guidelines, such as those employed by NASA's Jet Propulsion Laboratory [22], attempt to circumvent the most severe issues. Yodaiken [63] argues that the semantics of ISO C have drifted to the point of being unsuitable for systems programming, so these design problems cannot be patched by coding guidelines alone.

Governments and security agencies have emphasized memory safety as a strategic lever for reducing widely exploited vulnerability classes: advisories from the National Security Agency [10] and the Cybersecurity and Infrastructure Security Agency [64], and the National Cybersecurity Strategy's incorporation of memory-safe language principles [11], all reflect this direction. The implications are amplified in flight software, where constrained patchability, the ability of software compromise to alter spacecraft behavior, and shared-address-space execution on C/C++ stacks combine to turn a single memory corruption into a system-wide compromise [13].

2.5.2 Rust for Aerospace

Systems programming languages provide access to low-level computing and hardware functionality to meet the timing and resource-management requirements of embedded systems such as spacecraft. Snavely et al. [65] survey the history of systems-programming languages used in flight software (C, C++, Ada, Java, and Rust) and note that aerospace adoption of safer alternatives has been slow. In safety-critical aerospace, Ada and its formally verifiable subset SPARK have been the established choice for decades, with mature DO-178C toolchain support and strong adoption in avionics [51]. Rust offers a different set of mechanisms—compile-time safety through ownership and borrowing rather than contract-based annotation, an open-source ecosystem, zero-cost abstractions, and `no_std` support (omitting the standard library for bare-metal targets)—with the Ferrocene project providing a qualified toolchain for safety-critical certification [66]. The two approaches differ structurally as well as ergonomically. SPARK treats verification as a first-class language concern: pre/postcondition contracts, range subtypes, and the Ravenscar tasking profile are part of the language and its profiles, and off-proof bodies retain proof-driven specifications that callers can still rely on. Rust treats assurance as a composable concern: safe code discharges a baseline of memory-safety and data-race properties at compile time, while contract-style invariants and deductive or bounded proofs are supplied by external verification tools such as Kani and Verus. Each topology has consequences for assurance: SPARK's tightly integrated stack reduces per-property tooling decisions; Rust's compositional stack allows the architecture to choose where each property is enforced (language, type system, runtime, or proof tool).

Spacecraft impose precisely the constraints under which a systems language's safety-versus-performance tradeoff matters most: real-time deadlines, resource-limited processors, `no_std` environments, and the impossibility of post-deployment patching. Early efforts to use Rust on satellite software show feasibility and surface practical challenges. Projects such as `sat-rs` demonstrate early patterns for structuring satellite software in Rust, including abstractions for common mission behaviors and interfaces [67], and recent research examines how Rust's

guarantees translate to safety-critical aerospace contexts, including the interaction with real-time constraints, tooling, and certification evidence [51].

Adoption barriers include heritage-driven risk management, toolchain qualification and certification concerns, limited institutional experience, and integration challenges with existing codebases and interfaces [11]; Ferrocene addresses toolchain qualification directly. In practice, adoption in space systems is rarely a full rewrite. A more defensible path is to introduce memory-safe components where they provide the most leverage (for example, at parsing boundaries and message-handling interfaces) and to pair language adoption with coding standards and adversarial testing practices that explicitly target memory safety.

2.6 Summary

Flight software is a layered control stack — operating system, framework, mission applications — whose mission profile makes cyber compromise a physical-consequence concern rather than a data-integrity one. The broader space-cyber landscape shows why adversarial assumptions can no longer be treated as peripheral, and cyber resilience supplies the engineering frame for that concern: anticipate, withstand, recover, and adapt are NIST-grounded resilience functions that translate into architectural mechanisms rather than aspirations. cFS and F’ serve as representative open-source frameworks for concrete reasoning about the trust model that contemporary flight software inherits in operational practice. Memory-safe systems programming, with Rust as a candidate language, addresses one structural source of compromise that conventional implementation languages do not.

Chapter 3 turns from background to analysis by decomposing a representative flight software stack and identifying inherited attack surfaces. Chapters 4 and 5 then translate that analysis into testable requirements and architectural mechanisms.

Chapter 3

Threat Analysis and Attack Surface

Chapter Abstract. Representative flight software stacks expose attacker-controllable inputs and post-exploitation primitives below the mission-application layer. A bottom-up analysis of the cFS/RTEMS reference stack shows how inherited and emergent attack surface appears from the real-time operating system upward through the platform abstraction layer to mission applications. The result is a threat-informed account of how platform configuration, abstraction-layer complexity, permissive defaults, and memory-unsafe implementation choices shape attacker leverage before mission-specific application logic is considered.

Chapter Contributions. Three contributions follow. First, it establishes a flight-software-specific attack-surface methodology that begins from the lowest software layers (RTOS, board support package (BSP), abstraction libraries) rather than from a list of adversary techniques, making inherited and configuration-dependent risk visible in places typically treated as "given" in heritage reuse [13]. Second, it applies the methodology to a representative open-source stack (cFS on RTEMS) and produces a structured map of inherited and emergent vulnerability classes, including five structural pressure points that compound rather than compose independently. Third, it links those findings to mitigation leverage points that subsequent chapters convert into requirements (Chapter 4) and architectural mechanisms (Chapter 5).

Within the resilience arc developed in Chapter 1, this chapter supplies the *anticipate* step of NIST SP 800-160 Vol. 2: before requirements or architectural mechanisms can be justified, the analysis must show where an adversary can plausibly gain leverage [8]. The methodology and findings draw from an attack surface analysis of the cFS/RTEMS stack [13].

The key methodological choice is a *bottom-up* lens. Rather than beginning with a list of adversary techniques and searching for mitigations, the analysis starts from the lowest software layers that mediate all higher-level behavior: the real-time operating system, the board support

package configuration, and the abstraction libraries that connect flight applications to the platform. This makes inherited attack surface and configuration-dependent risk visible in places often treated as “given” in mission software reuse.

3.1 Related Work

Three strands of prior work inform the analysis. The first examines complexity as adversary terrain in space computing. The second is attack-surface mapping as it has developed in adjacent high-consequence domains—embedded CPS, automotive, aviation, and industrial control systems. The third is the space-specific threat taxonomy, SPARTA, which connects concrete findings to recognizable adversary behaviors.

3.1.1 Complexity in Spacecraft Computing Systems

Complexity growth in spacecraft computing directly enlarges the cybersecurity problem. Spacecraft are evolving from single-board-computer architectures into distributed computing systems with varied hardware, operating systems, and application software. Decreasing costs and size of compute and storage resources drive this shift. Proliferated constellations—dozens to thousands of interconnected spacecraft—introduce additional distributed-system complexities, complicating secure system design and giving adversaries room to operate undetected [13].

Modern spacecraft range from single-board microprocessor designs running an RTOS to distributed systems with multiple computing subsystems (radios, guidance systems, power management, and separate hosted payloads), each with its own software environment. Some architectures have shifted toward a distributed model in which the spacecraft bus is minimal and the payloads carry most of the computational capability, multiplying the interfaces and internal networks that link subsystems.

Complexity does not itself diminish resilience, but it obscures which resilience properties actually hold and where risk sits. NIST Fellow Dr. Ron Ross states the consequence directly

[68]:

> The real cyberwar is being fought on the field of complexity. [...] It will take a bare-knuckled, pound it out on the ground a yard at a time, systems and security engineering approach—applying rigorous design principles and architectures that minimize complexity and maximize assurance and trust. If you cede control of critical components such as operating systems to adversaries by failing to address complexity and assurance, they will use subversion to own the cyber battle space and turn your high-tech into no-tech.

Ross's argument is that complexity itself is the operationally decisive terrain. The bottom-up methodology follows from that: examine what the operating system and its abstraction layers actually expose, rather than start from frameworks or controls.

3.1.2 Mapping a Spacecraft's Attack Surface

Mapping attack surface in a spacecraft is not the same problem as mapping it in an enterprise or industrial control system. Thummala et al. [25] give a systematic treatment of how orbital mechanics couple directly into the resilience model: contact geometry bounds when interactive security protocols can execute, radiation-induced bit flips corrupt cryptographic state in ways that mimic or mask cyber attacks (see their Table I), and tight subsystem interdependence turns localized compromise into cascading mission failure. Their seven-constraint framework strengthens the argument that space cybersecurity requires domain-specific design rather than adaptation of enterprise or industrial control system practices.

An *attack surface* encompasses the various ways an attacker can access, influence, or communicate with a system [18]. Defending a system begins with understanding that surface from both offensive and defensive perspectives. For flight software, the attack surface extends well beyond the radio uplink and ground interfaces to include internal control channels (software buses, inter-process communication (IPC), file systems), platform configuration choices, and

inherited capabilities of embedded operating systems. The practical security question is therefore not “does the system use encryption,” but “what behaviors can be triggered if an attacker reaches a given interface, and how far can they move from there?”

Methodologies for space-system attack-surface analysis have begun to appear, adapting established threat-analysis approaches such as Microsoft’s Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege (STRIDE) [69] and the Process for Attack Simulation and Threat Analysis (PASTA) [70] to space-specific constraints. Space-specific threat frameworks such as SPARTA provide structured taxonomies for this purpose.

Detailed attack surface studies of spacecraft flight software remain scarce in public discourse. Reports on cyber incidents such as the ViaSat attack [7] and the CYSAT demonstration [44] offer high-level insights but lack the granular technical detail needed to identify specific architectural vulnerabilities in flight software.

Related disciplines that share the same constraints (embedded microprocessors, real-time deadlines, and physical consequences of failure) have developed their own attack-surface methodologies. In embedded and cyber-physical systems (CPS), Papp et al. [71] survey threats and propose a taxonomy that mirrors the tactic-level organization later adopted by SPARTA, and Easwaran et al. [72] extend the analysis to real-time scheduling concerns. The automotive sector has produced secure-design frameworks and vulnerability studies across successive generations of in-vehicle networking: Sagstetter et al. [73] frame the architectural security challenges for heterogeneous automotive hardware/software stacks, Edwards et al. [74] apply MISRA and CERT C static analysis to identify latent vulnerabilities in production control-unit software, and Zhang and Olmsted [75] examine how operating-system attack surfaces intersect with the functional-safety requirements of autonomous vehicles. Aviation researchers have followed a similar trajectory: Kiesling et al. [76] propose a model-based risk-assessment method for aviation cybersecurity, and Habler et al. [77] give a comprehensive aircraft-system taxonomy organized by MITRE ATT&CK tactics and STRIDE categories. Industrial control and supervisory control and data acquisition (SCADA) systems are the most mature adjacent domain, with

Cybersecurity and Infrastructure Security Agency (CISA) operational guidance [78] and a dedicated MITRE ATT&CK for ICS framework [48] providing structured threat intelligence.

Bottom-up analysis has a long history in other high-consequence domains. Passmore and Ignatovich [79] argue that behavioral reasoning about a system requires verified properties of the supporting subsystems that execute its intentions, a principle that extends naturally to flight software stacks. The automotive and aviation literature similarly emphasizes that complexity, interface sprawl, and opaque middleware can undermine safety and reliability assurance when adversarial manipulation is not factored alongside failure modes.

3.1.3 SPARTA as the Space Threat Taxonomy

SPARTA provides a space-specific taxonomy of adversary techniques and countermeasures organized by tactic categories: reconnaissance, initial access, execution, lateral movement, persistence, defense evasion, exfiltration, and impact [47]. Each technique instance carries a Normalized Risk Score (NRS) on a 0–25 scale that weights likelihood against consequence, enabling prioritized analysis. SPARTA also provides an Indicators of Behavior (IOB) framework—version 3.1 at the time of this analysis, with 193 indicators across 10 categories—that complements the technique taxonomy by providing detection and monitoring patterns for identifying adversary activity in operational systems. The bottom-up analysis uses SPARTA to connect concrete findings to recognizable adversary behaviors. A complementary requirements-focused framework, HAT TRICK, appears in Chapter 4, where its role in bounding the requirements space is developed.

3.2 Methodology: Bottom-Up Attack Surface Analysis

3.2.1 Why a Bottom-Up Lens Matters

Top-down threat models are necessary for prioritization, but they often assume the existence of clean trust boundaries and well-understood interface contracts. Flight software stacks rarely

provide those by default. In practice, reuse-driven architectures combine mission applications, middleware services, abstraction layers, and RTOS capabilities into a single linked image with extensive implicit trust. When that is the baseline, a bottom-up analysis becomes a practical prerequisite: it reveals which low-level behaviors are inherited, which interfaces are actually exposed, and which configuration choices quietly expand the reachable surface area [13].

This approach draws inspiration from the principle that one cannot properly reason about the possible behaviors of a system higher in the stack unless one has verified the relevant properties of the supporting subsystems that will be executing its intentions [79]. In the context of flight software, this means the RTOS is the correct starting point for any meaningful attack-surface analysis: the rest of the flight software system completely depends on its behavior. Any attack on the RTOS can affect the overlying system, and manipulation of the RTOS may not be detectable at higher layers. Once the lower layers of the stack are examined and weaknesses mapped, one can put mitigations in place and turn attention to higher layers, which carry their own attack surface.

3.2.2 Selection of Representative System

A representative flight software stack is needed to map attack surface, evaluate analysis methodology, and identify mitigation strategies concretely. Many spacecraft systems run embedded platforms with a Real-Time Operating System (RTOS); others run conventional platforms with operating systems like Linux. Analysis of an RTOS/embedded platform establishes the foundational patterns—inherited capabilities, abstraction boundaries, default configurations—that any flight software stack inherits from its underlying OS. Non-embedded platforms carry additional software complexity and warrant separate study, but the bottom-up methodology transfers to them.

The criteria for selecting a representative system, in no particular order of priority, include:

- Broad use across actual space missions, grounding the study in real-world space requirements and constraints.

- Public accessibility for in-depth analysis down to the source code level.
- Broad coverage of spacecraft operations, from core hardware management to mission-specific tasks, to encompass a wide range of software use cases.

On these criteria, cFS running on RTEMS is the representative stack [13]. The cFS layered architecture is described in Chapter 2 (Section 2.4.1; see also Figure 2.1): cFS composes mission applications, core executive services, an OSAL, and the underlying RTOS into a single linked image with extensive implicit trust between layers.

The OSAL mediates all interaction between cFS and the operating system through wrapper code and BSP configuration. At the time of this analysis, OSAL supported RTEMS 5.x, with RTEMS 6.x support in progress. RTEMS 5.3 is the primary reference for the analysis unless otherwise specified. The OSAL interfaces exclusively through the RTEMS "classic" application programming interface (API), and the RTEMS BSP manages build-time configuration and module inclusion.

3.2.3 Analysis Framework

The representative system spans multiple software layers, from application-level services to platform support. Rather than dissecting each layer uniformly, the analysis concentrates on the foundational elements: the Real-Time Operating System and the API infrastructure in the abstraction layer above it. The RTOS controls hardware interactions, memory allocation, and scheduling—functions that, if compromised, cascade into every layer above [13].

The analysis uses several mutually reinforcing lenses:

- **Dependency analysis** identifies inherited functionality, and therefore inherited attack surface, introduced by library and platform choices. Dependency analysis explains the interrelations within system components, identifying those critical for overall functionality. With this information in hand, components can be prioritized for enhancing cyber resilience.

- **Critical-path analysis** scrutinizes essential code paths, focusing attention on code an attacker would target to influence command execution, telemetry integrity, or fault management behavior.
- **Data-flow and interface analysis** examines data movement and component communication, following untrusted inputs as they traverse parsing, routing, and storage mechanisms to identify where validation assumptions matter.
- **Traditional vulnerability and code analysis** seeks language usage flaws, improper function calls, and data management issues inherent to the implementation language and environment.

The analysis prioritizes dependency and critical-path methods as foundational investigation tools, guiding further examination with the other lenses. The focus remains on the system's lowest layers, primarily the RTOS core, its board support package, and the cFS abstraction layer's OSAL and Platform Support Package (PSP). Analyzing the interactions between OSAL components, RTEMS managers, and device drivers sheds light on their connectivity, relationships, and standard configurations [13].

In practice, the analysis proceeded as follows. Source code for the public cFS, OSAL, and RTEMS 5.3 distributions was obtained from their public repositories. The baseline was the publicly documented stack as inherited by a new mission team: default OSAL/RTEMS integration assumptions, build configuration files, and the BSP options compiled into a representative mission image unless a mission-specific hardening step was explicit in the public materials. Dependency analysis was conducted through manual inspection of build configuration files (`CMakeLists.txt`, BSP configuration headers) and by tracing `#include` chains and linker inputs to enumerate which RTEMS modules, file systems, and services are compiled into that baseline image.

Critical-path analysis followed the command ingestion path from the cFS Software Bus receive call through OSAL wrappers to RTEMS message queue primitives, identifying each

layer where validation or authority enforcement could (but does not) occur. Data-flow analysis traced untrusted inputs (uplinked commands, sensor data) through parsing, routing, and storage to identify where validation gaps exist. Code-level vulnerability analysis focused on C-language patterns known to produce memory safety issues (unchecked buffer operations, pointer arithmetic across trust boundaries) in the OSAL wrapper code and RTEMS shell implementation. No automated static analysis tools were applied in this phase; the analysis relied on manual code review informed by the SPARTA technique catalog to prioritize which code paths to examine and is therefore best understood as a reproducible analyst workflow rather than as an exhaustive vulnerability census.

The emphasis is intentionally on these lowest layers because that is where configuration often trades security for developer convenience, and where compromise can collapse into system-wide impact.

3.3 Attack Surface Findings in cFS on RTEMS

cFS is designed to be customized by individual missions; the permissive defaults documented in the public distribution are starting points, not prescriptions. Public defaults are the baseline for this analysis: the findings characterize the stack as a new mission team would inherit it before mission-specific hardening. Missions that harden defaults responsibly can eliminate specific surfaces identified here; the analysis captures the failure mode that appears when inherited platform assumptions are accepted without scrutiny.

3.3.1 OSAL Complexity and Opacity

The OSAL in the core Flight System provides a uniform API across multiple operating systems (RTEMS, VxWorks, and Linux), enabling mission reuse, rapid prototyping on Linux, and standardized expression of cFS's OS requirements [13].

NIST SP 800-160 Vol. 2 emphasizes minimizing system complexity and redundant code as a

design principle for resilience [8, p. 102]. The OSAL works against that principle by interposing a layer of wrapper code between cFS applications and the target OS. Each wrapper must reconcile mismatched APIs and sometimes supply “missing” behavior expected by cFS, so the wrappers are not thin adapters; they contain non-trivial logic that can introduce its own defects. By obscuring direct interactions with the underlying system, this indirection also complicates the tuning of platform security controls and the detection of malicious activity within cFS [13]. The design pressure from a resilience perspective is therefore toward simplifying the abstraction boundary, narrowing the surface area of OS services consumed by flight software, and making privilege boundaries explicit rather than implicit.

3.3.2 RTEMS Execution Model and Post-Exploitation Primitives

RTEMS exhibits a fairly standard architecture for a Real-Time Operating System, characterized by its “real-time executive” that adopts a single process architecture [80]. In this model, all code shares a unified address space and is statically linked, except for the instances using RTEMS’ dynamic loader. RTEMS allows task management through two distinct execution APIs: Classic or POSIX. However, because of its singular process architecture, RTEMS does not offer memory protection or security isolation among tasks, leading to a combined, statically linked binary image for the RTOS, API, and applications [13].

In that model, memory safety faults and privilege escalations are not naturally contained. A compromise that reaches arbitrary memory write capability can rapidly become system-wide, including the ability to influence command processing, telemetry reporting, and fault management behavior. There are no hardware-enforced walls between a compromised application task and the RTOS scheduler or the command-processing service. Vulnerabilities in the comparable FreeRTOS kernel that led to remote code execution, denial of service, and data leakage illustrate the practical consequences of this architecture [81].

RTEMS also supports dynamic loading of executable code and data (object files) [82], integrating loaded modules into the same flat address space as statically linked code. The

RTEMS developers themselves caution against using the dynamic loader in real-time code, citing lack of protections and potential for misuse. Under an adversarial model, dynamic loading is an escalation primitive: if an attacker can reach the loader interface (directly or via an exposed shell), they can introduce new code into the running image. Even if an operational deployment disables dynamic loading, the presence of the code paths and configuration switches highlights a recurring theme: embedded platforms often ship with powerful capabilities that must be explicitly audited out of mission builds to avoid expanding the reachable surface [13].

Developer conveniences compound the single-address-space risk. RTEMS incorporates a Unix-style shell for file manipulation, system introspection, memory dumping and editing, and dynamic loader access [83], and the OSAL activates it within its RTEMS BSP configuration by default, permitting the full command set [13]. In a flat-memory RTOS with no privilege separation, an attacker who reaches the shell inherits its full capabilities and can use legitimate built-in tools rather than delivering external payloads—the “living off the land” technique now common in cyber operations [84]. In mission builds, an enabled shell is an attack-surface risk unless it is removed or gated by mission-appropriate authentication and authorization.

3.3.3 BSP Configuration and Permissive Defaults

RTEMS provides hardware-specific support and device drivers through board support packages. The OSAL’s configuration of the RTEMS BSP, outlined in the `bsp/pc-rtems/src/bsp_start.c` file, is executed using RTEMS-defined parameters. The default setup merits scrutiny from a cyber resilience perspective [13].

The default BSP configuration supports four file systems: the In-Memory File System (ImFS), DOS (FAT) file system, device file system (DevFS), and RTEMS File System (RFS), with ImFS serving as the default. The configuration does not disable any ImFS functionalities, though such restrictions are possible. Board support packages are frequently configured to maximize developer convenience: multiple file systems are enabled, permissive defaults are selected, and debugging interfaces are left available.

These defaults matter even if a mission "does not use" a given capability. Code compiled into a monolithic image is still part of the reachable surface after exploitation: unused drivers and file system code remain in memory and can serve as vectors for persistence, data corruption, or system crashes triggered by alteration of state beneath an active task.

For mission builds, BSP configuration is an attack-surface reduction exercise: unused file systems and services are removed, in-memory storage capabilities are restricted where feasible, and remaining administrative interfaces are gated by explicit authentication and mission-appropriate authorization policies. Where such controls are not feasible on the platform, the limitation becomes a constraint on higher-level architectural containment.

3.3.4 Memory Safety as an Attack-Surface Multiplier

In C-based flight software stacks, memory safety vulnerabilities (buffer overflows, use-after-free errors, and related classes) multiply the effective attack surface because they convert ordinary interfaces into code execution pathways. In systems that lack strong isolation, the amplification is acute: a single memory corruption in a message parser or filesystem API can become arbitrary behavior at the process level.

Mitigations include adoption of memory-safe languages, static and dynamic analysis tooling for existing C codebases, and incorporation of memory safety into coding standards and adversarial testing regimes; Chapter 2 discusses these approaches. The finding here is that memory unsafety is an attack-surface multiplier: it amplifies every other surface identified in this analysis.

3.4 From Surfaces to Techniques: Threat Mapping

Attack-surface findings become useful to design only when they are expressed in a threat vocabulary that connects software surfaces to recognizable adversary behavior. Threat assumptions, representative SPARTA techniques, and multi-step attack chains together demand authority

enforcement, isolation, and observability from any mitigation — the requirement categories Chapter 4 develops.

3.4.1 Threat Actors and Assumptions

The primary adversary model is a high-capability remote attacker (for example, a nation-state actor) capable of exploiting zero-day vulnerabilities, manipulating command and telemetry flows, and persisting across long mission timelines [3]. Lower-capability adversaries (criminal, insider, opportunistic) remain relevant; designing for the high end produces trust boundaries and recovery behavior that also hold under weaker threats.

Industrial control and SCADA systems provide an instructive analogue. These systems share spaceflight's constraints: long service lifetimes, limited patch cadence, and tight coupling between cyber compromise and physical effects. ICS security practice treats configuration, protocol gateways, and operational dependencies as first-class attack surface rather than as incidental implementation details, as reflected in CISA's operational guidance [78] and in MITRE's ATT&CK for ICS technique catalog [48]. Adopting the same stance for flight software, treating every default-enabled capability as a potential adversary primitive, is the methodological contribution of the bottom-up approach.

3.4.2 Mapping Representative Techniques to Flight-Software Surfaces

SPARTA organizes adversary techniques by tactic prefix — EX (Execution), DE (Defense Evasion), IA (Initial Access), PER (Persistence), among others — with numeric IDs and dotted sub-IDs for specific variants. Several technique families recur across flight software stacks and map directly to the surfaces identified in Section 3.3:

- **Command and data manipulation.** Surfaces include uplink parsing, internal message routing, and telemetry packaging. Relevant SPARTA techniques include EX-0009 (Control Telemetry Data, NRS 25), EX-0013.01 and EX-0013.02 (Modify On-Board Values for

stored and volatile commands, both NRS 25), EX-0014.01 (Spoof Time, NRS 25), and EX-0014.02 (Spoof Bus Traffic, NRS 25). The resilience requirement is not only cryptographic integrity on the link, but validation of semantics and authority at the point where commands become behavior. In the single-address-space RTEMS model, a compromised task can reach these pathways without requiring privilege escalation in the traditional sense.

- **Execution of malicious or valid programs.** Surfaces include dynamic loading capabilities, scripting or shell interfaces, and any mechanism that allows code or configuration changes at runtime. SPARTA techniques include EX-0001.01 (Compromise Boot Memory—Flight Software, NRS 25), IA-0007.02 (Malicious Commanding via Valid Ground Station, NRS 25), and EX-0009.01 (Exploit Code Flaws in Flight Software, NRS 25). The RTEMS shell and dynamic loader are the most direct examples identified in Section 3.3; both represent legitimate development facilities that become adversary capabilities if left enabled in operational builds.

- **Disabling or subverting fault management.** Surfaces include health-monitoring interfaces and any shared-state assumptions that allow one component to influence a peer component's fault handling. Relevant techniques include DE-0001 (Disable Fault Management, NRS 24), EX-0012.10 (Modify C&DH Subsystem, NRS 24), and EX-0012.11 (Modify Watchdog Timer, NRS 24). In a flat single-process RTOS model, there is no enforcement boundary preventing a compromised task from altering the state that fault managers observe.

- **Persistence and stealth.** File system interfaces, such as the ImFS and writable filesystems enabled by default, provide a mechanism for an attacker to persist code or modified configuration across software resets if those storage areas are not integrity-checked at boot. SPARTA maps this to PER-0002.02 (Software Backdoor, NRS 24), PER-0003 (Ground System Presence, NRS 25), and DE-0007 (Evasion via Rootkit, NRS 24).

Nine of the mapped technique instances carry the maximum NRS of 25, indicating the

highest-consequence risks in the SPARTA framework. Their concentration in the command manipulation and execution categories drives the priority of command authentication, authority enforcement, and isolation as foundational resilience requirements. The mapping is intentionally conservative: it shows that the representative attack surfaces identified in the public/default cFS/RTEMS baseline correspond to recognizable adversary behaviors, not that every SPARTA technique is equally likely in every mission.

3.4.3 Attack Chain Archetypes

Individual techniques rarely occur in isolation. A realistic adversary chains multiple techniques into a multi-step progression that moves from initial access through execution and persistence to mission impact. Three representative kill chains illustrate how selected surfaces identified in Section 3.3 can combine into end-to-end attack progressions against flight software stacks:

1. **Ground-to-flight command injection.** An adversary compromises a ground station or exploits a valid command session (IA-0007.02), then injects commands that modify on-board values (EX-0013.01) or alter system configuration (EX-0014.02). If the RTOS provides no authority enforcement beyond link-layer authentication, the attacker can escalate from command injection to persistent control by writing to boot memory (EX-0001.01) and suppressing telemetry evidence (EX-0009.01). In the cFS/RTEMS stack, this chain is viable because the Software Bus imposes no publish-side restrictions: any application can send any message identifier.

2. **Supply chain compromise.** An adversary introduces malicious code during the build process, whether through a compromised dependency, a tampered toolchain, or a modified board support package (IA-0001.02). The malicious code is compiled into the monolithic flight image and persists across resets (PER-0002.02). At runtime, the code exploits the single-address-space model to modify telemetry data (EX-0009.01) or disable fault management (DE-0001), while evading detection by operating within the legitimate execution

context. This chain exploits the absence of binary attestation and the flat memory model characteristic of traditional RTOS architectures.

3. **Hosted payload lateral movement.** A compromised hosted payload, whether through supply chain compromise or exploitation of a payload-specific interface, attempts to pivot to flight-critical services (LM-0001, LM-0002). Hosted payloads are often separate hardware and software elements, so the relevant boundary is the command, telemetry, and data interface between the payload and the spacecraft bus rather than a shared process address space. If that interface is implicitly trusted—whether implemented as a hardware bus, a software bus, or a bridge between them—the attacker may be able to inject messages, influence command routing (EX-0014.02), interfere with fault-management inputs (DE-0001), or exfiltrate data through telemetry channels (EXF-0003.02). This chain applies directly to commercial missions that host third-party payloads with less rigorous software assurance than the primary flight software.

Two patterns emerge across the three chains. Single-address-space execution enables monolithic compromise, and unauthenticated or implicitly trusted buses enable lateral movement across separately hosted components. Permissive RTOS/BSP defaults amplify both.

The cFS Software Bus deserves specific attention as a structural enabler because it mediates all inter-application communication and enforces none of the properties that containment requires. Four distinct threat classes arise from its design:

- **Unauthorized read.** Any application can subscribe to any message identifier without restriction.
- **Unauthorized publish.** Any application can publish any message identifier, enabling command injection from a compromised non-critical service.
- **Sender identity spoofing.** The `APP_ID` and `PROCESSOR_ID` fields in message headers are self-reported by the sending application, with no bus-level validation, so a compromised component can impersonate any other.

- **Message flooding.** The Software Bus provides no rate limiting, no per-publisher anomaly detection, and no signal to downstream consumers when queue overflow causes silent message drops.

These are not implementation bugs; they are consequences of a cooperative trust model in which all applications are assumed to be correct. Under an adversarial model, the Software Bus amplifies rather than contains single-component compromise. These architectural gaps, rather than any single vulnerability, are what allow individual technique instances to compose into system-wide compromise.

3.5 Summary

A bottom-up analysis of cFS on RTEMS reveals several recurring pressure points: abstraction-induced complexity in the OSAL, weak isolation in single-address-space execution models, developer conveniences that become post-exploitation primitives (most prominently the RTEMS shell and dynamic loader), permissive default BSP configuration that expands the reachable surface, and memory unsafety that converts ordinary interfaces into control pathways [13]. The findings connect to recognizable adversary behaviors in SPARTA: the RTEMS shell maps to living-off-the-land execution; the single-address-space model maps to privilege escalation and lateral movement after initial compromise; permissive file system defaults map to persistence mechanisms; and memory safety weaknesses map to the broad class of code injection and memory corruption techniques.

Taken together, these findings show that cyber resilience for flight software must begin with a concrete understanding of attack surface as it exists in real stacks. The analysis explains why Chapter 4 centers on authority and privilege, interface integrity, isolation and containment, configuration hardening, boot integrity, language-level safety, and monitoring/response. These are not generic control categories; they follow from the concrete ways compromise propagates through cFS/RTEMS: implicit internal trust, shared execution context, permissive platform

defaults, developer-facing post-exploitation primitives, and memory-unsafe implementation paths.

Chapter 4

Cyber Resilience Requirements Derivation

Chapter Abstract. Threat analysis becomes design-time obligation through a repeatable requirements-derivation process. Requirements form the bridge between attack-surface findings and enforceable architecture, using secure-by-component design and adversary-informed scoping to keep the resulting statements traceable. Mission-level concerns such as permanent loss of control can be narrowed into component-local obligations that later architecture and evaluation work can test directly.

Chapter Contributions. Three contributions follow. First, it establishes a minimum-requirements methodology grounded in permanent loss-of-control scenarios rather than in generic checklists, making the "what must be true" question concrete in the flight-software context [14]. Second, it adapts the IEEE P3349 secure-by-component workflow to flight software with the HAT TRICK threat framework, connecting threats to components, components to design approaches, and design approaches to concrete shall-statements with explicit traceability [15]. Third, it produces a testable requirement set that is traceable to threats and to mission consequences, suitable for later architectural and evaluation work without per-mission re-derivation.

Resilience claims that remain aspirations are not useful. Requirements must specify what the system must do under adversarial pressure and how that behavior can be verified — which is what the threat analysis of Chapter 3 now needs to be translated into.

The approach builds on prior work that established a minimum set of secure-by-design approaches for space systems [14] and on the secure-by-component design philosophy developed for the Institute of Electrical and Electronics Engineers (IEEE) P3349 standard [85]. Those foundations are refined here into *testable flight software requirements* using a secure-by-component workflow augmented by the HAT TRICK threat framework [15]. The result is not a

universal checklist. It is a repeatable derivation process that produces requirements with explicit traceability to threats and to mission consequences.

4.1 Related Work

The derivation method draws on four areas of prior work: the general practice of security requirements as a distinct engineering artifact; the existing space cybersecurity standards landscape and the flight-software gap within it; the IEEE P3349 secure-by-component standard that directly informs the workflow; and threat-informed requirements techniques (attack trees, fault trees) that shape the approach.

4.1.1 Security Requirements as an Engineering Artifact

Requirements engineering in high-consequence systems is inseparable from assurance: requirements are intended to be traced, verified, and used to structure both development and test. NIST SP 800-160 Vol. 1, introduced in Section 2.3 as part of the trustworthy systems engineering framework, formalizes this idea for systems security engineering: trustworthiness must be engineered across the lifecycle, and evidence depends on explicit requirements rather than on post hoc justification [52].

The key difference between safety-centered requirements and adversarial requirements is not form (both often use “shall” statements) but the failure model they encode. Safety practice often assumes random faults, bounded environmental disturbances, and predictable recovery sequences. In contrast, cyber threats involve an intelligent adversary selecting inputs and execution paths to defeat defenses, including attempts to subvert monitoring and recovery logic. As a result, cyber requirements must explicitly constrain authority, validate inputs at trust boundaries, and define acceptable degraded modes in the presence of deception.

General-purpose control catalogs such as NIST SP 800-53 [86] enumerate broad technical and procedural controls but are not calibrated to space-specific adversary behavior or to the

degraded observability and recovery conditions of on-orbit systems. Compliance with a generic catalog does not, on its own, produce requirements that account for an adversary who adapts once inside the vehicle boundary.

4.1.2 Space Cybersecurity Standards and the FSW Requirements Gap

Existing standards and policy guidance such as SPD-5 address top-down risk governance, and CCSDS Space Data Link Security (SDLS) specifies cryptographic protections for specific communication layers [87, 88]; neither defines what on-board software must do when prevention fails or when compromise occurs within the vehicle boundary [14].

Viswanathan et al. [85] articulate five limitations in current approaches that bear on requirements derivation:

- Most frameworks target post-deployment risk management by operators rather than proactive threat mitigation by developers.
- Existing guidance is often too abstract for application to space system development.
- Non-technical guidance predominates and does not support technical security implementation.
- Guidance is siloed by mission segment due to the fragmented nature of space systems.
- The lack of space-specific standards forces adaptation of non-space frameworks, producing suboptimal results.

This gap motivates bottom-up requirements work. The Office of the National Cyber Director has argued that measurable software security depends on identifying and enforcing low-level building blocks rather than relying exclusively on process-based assurance, with an emphasis on space systems [11]. The IEEE P3349 standard (Section 4.1.3) takes this approach for space systems. What remains challenging in practice is determining which subset of approaches

and requirements are “minimum” for a stated mission objective, and how to convert high-level approaches into testable flight software obligations.

4.1.3 The Secure-by-Component Standard for Space Systems

The IEEE Standard for Space System Cybersecurity (Working Group P3349) addressed the foregoing limitations by developing an international standard for space system cybersecurity centered on a secure-by-component design philosophy [42]. The standard did not seek to replace other standards; instead, where possible, it pointed to and referenced existing work by bodies such as German Federal Office for Information Security (BSI), European Cooperation for Space Standardization (ECSS), NIST, and CCSDS when their granularity suited the required bottom-up secure-by-design approach [14].

The secure-by-component approach moves away from categorizing missions into classes or recommending a uniform reference architecture. Instead, it adopts a system-of-systems perspective when considering secure-by-design for space missions, given that each segment of a space system can be decomposed into subsystems and further broken into components and subcomponents. The foundational components, referred to as *secure blocks*, can be combined to create secure architectures tailored to a given mission’s requirements [14].

The standard produces an extensive catalog of “shall” statement requirements for each subcomponent that can be assembled to formulate secure blocks for space missions. A systems engineer would build one of two versions of secure blocks: (a) a *maximum-security design*, crafted to mitigate all the identified threats with respect to the high-level objectives of a mission; or (b) a *minimum-security design*, aimed at achieving a narrowly defined security objective (Figure 4.1). These two levels allow system architects to choose the design that best fits the unique objectives of their space mission [14].

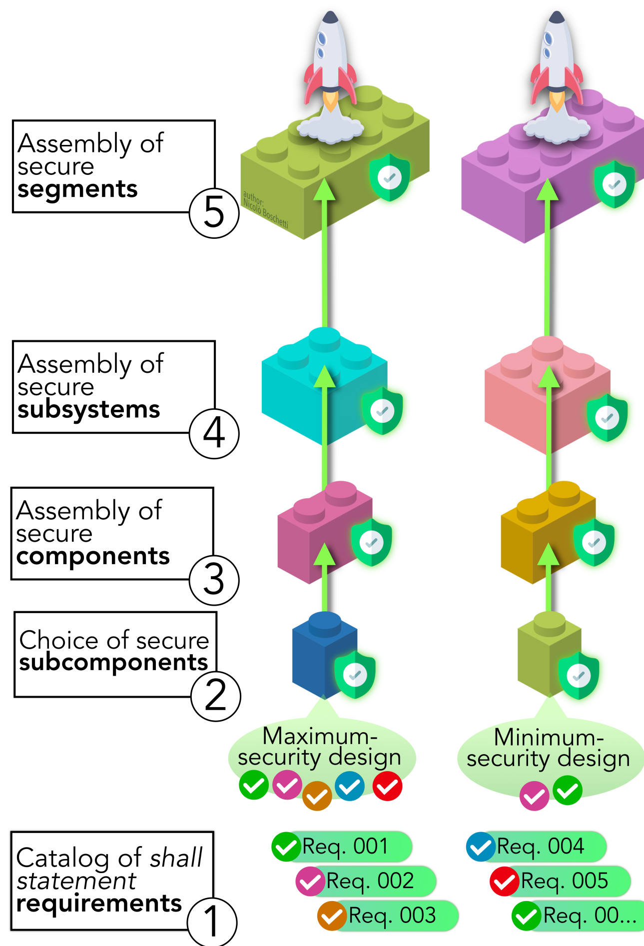


Figure 4.1 Secure-by-component assembly process. Requirements from a catalog of "shall" statements (Step 1) are composed into secure subcomponents (Step 2), components (Step 3), subsystems (Step 4), and segments (Step 5) to produce a complete mission architecture. Maximum-security designs mitigate all identified threats; minimum-security designs target a narrowly defined objective. Adapted from Falco et al. [14].

4.1.4 Threat-Informed Requirements Approaches

Threat-informed requirements engineering grounds "shall" statements in concrete threat behavior and mission consequence rather than in abstract control catalogs. Attack trees decompose an adversary's goal into sub-goals and concrete actions, making pathway enumeration explicit but suffering a combinatorial problem: the permutations are largely subjective and span an arbitrary mix of confidentiality, integrity, and availability challenges [14]. STRIDE, the threat-categorization scheme used in the original secure-by-component workflow [85], is general-purpose rather than space-specific, which motivates the move to HAT TRICK in Section 4.2.3.

Fault-tree analysis takes a different angle. Rather than enumerating attacker actions, it selects a mission-level failure mode (for example, permanent loss of control) and decomposes how that failure could occur across mission segments and interfaces [14]. Compared with attack trees, fault trees reduce dependence on a specific adversary playbook and make "minimum" easier to define: the goal is to prevent a narrowly defined unacceptable outcome, not to enumerate every possible adversary path. A fault tree anchored to a mission-level failure mode produces a pointed and less subjective analysis (Figure 4.2) [14].

The workflow combines fault-tree-style outcome framing (via mission resilience priorities) with structured adversary characterization (via HAT TRICK and SPARTA).

4.2 Secure-by-Component Methodology

4.2.1 Method Overview

The approach builds on the secure-by-component design methodology [85], augmented by two complementary frameworks: NIST SP 800-160 for systems security engineering principles [8] and HAT TRICK for deriving cybersecurity requirements against high-tier adversaries [49]. The combination embeds security into the flight software development process from requirements definition through implementation rather than relying on post-deployment controls [15].

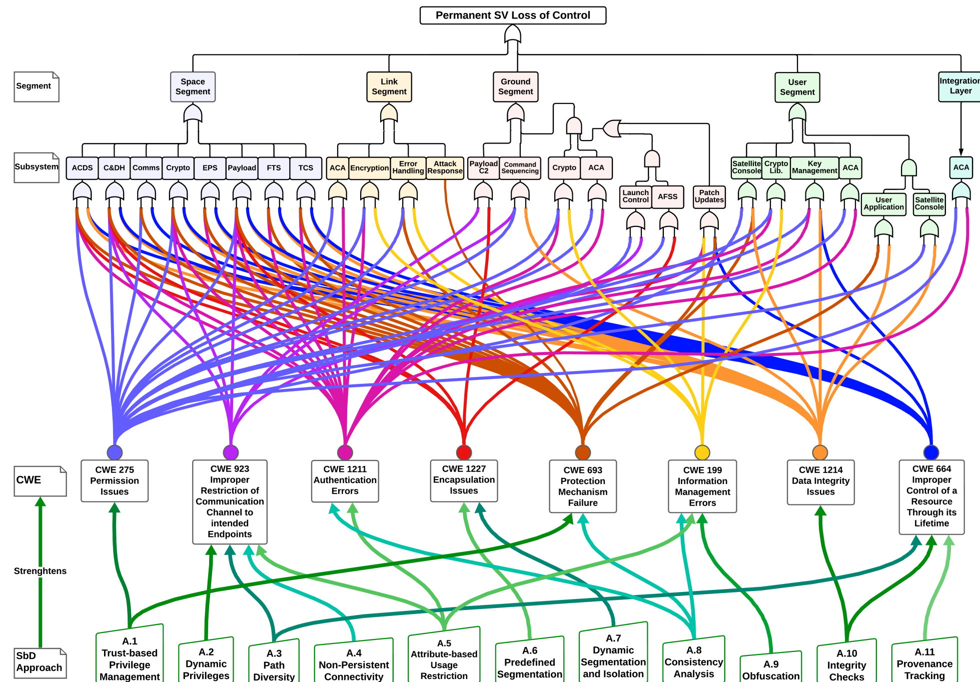


Figure 4.2 Example security fault tree anchored to a mission-level unacceptable outcome. Intermediate events decompose the outcome across segments and interfaces, and basic events identify specific preventable conditions. Adapted from Falco et al. [14].

The overall process follows eight steps that augment the six-step secure-by-component process [85] with framing and implementation activities [15]:

1. Leverage NIST secure systems engineering principles to guide the approach. *(Framing)*
2. Define mission resilience priorities to guide the development of cyber resilience requirements. *(Framing)*
3. Decompose the flight software architecture into components. *(Secure-by-component Step 1)*
4. Analyze the attack surface of each component. *(Secure-by-component Step 2)*
5. Select relevant SPARTA techniques for each component, using HAT TRICK threat events as a guide. *(Secure-by-component Step 3, modified)*

6. Identify secure-by-design principles to mitigate the selected SPARTA techniques. *(Secure-by-component Step 4)*
7. Define secure blocks and generate cybersecurity requirements for each. *(Secure-by-component Steps 5 and 6)*
8. Create a technical implementation plan that incorporates cybersecurity requirements by embedding cyber resilience-driven functional and technical specifications into the flight software architecture. *(Implementation)*

The principal modification to the baseline workflow of Viswanathan et al. [85] occurs at Step 5: HAT TRICK threat events replace STRIDE as the categorization scheme, and SPARTA techniques supply space-specific adversary behavior rather than generic threat types [49].

4.2.2 Framing the Requirements Methodology

NIST SP 800-160, whose two volumes address systems security engineering and cyber resiliency engineering respectively (see Section 2.3), provides the overarching framework [8]. Its key contribution here is the principle that security and resilience must be embedded across the system lifecycle, not applied as a separate assurance layer. That principle translates directly into the requirement that functional and technical "shall" statements be generated at design time and carried forward into test and evaluation plans [15].

Within that framework, this methodology prioritizes the integrity of spacecraft commanding as its mission resilience focus. A breach of command integrity could enable adversaries to issue malicious commands, disrupt operations, or damage the spacecraft, a concern underscored by NASA after alleged attacks on two of its satellites [15]. Alternative priorities (e.g., communication availability, data confidentiality) would follow the same derivation process with different emphasis.

4.2.3 Integrating the HAT TRICK Framework

Section 2.2.4 introduced HAT TRICK as a complementary, requirements-focused framework [49]. Its cyberspace threat matrix—three access vectors crossed with five threat events—offers a structured approach to bounding the adversary problem space. For requirements derivation, the key detail is how the framework converts that matrix into 53 requirements organized in three blocks [49]:

Block A — Cyberspace Threat Events (Requirements 1–25):

Requirements derived from the five threat events. They cover prevention, detection, reporting, neutralization, and isolation of unauthorized data (1–5); monitoring and reporting of anomalous behavior (6–7); prevention, detection, reporting, and neutralization of unauthorized programs (8–11) and unauthorized execution of authorized programs (12–15); prevention, detection, and reporting of denial of authorized data (16–18); and confidentiality protections against unauthorized data access (19–25).

Block B — Access Vectors (Requirements 26–45):

Requirements addressing resilience to compromise through each access vector: data connections and network access, supply-chain compromise, physical access to system components, and abuse by authorized subjects.

Block C — Type B Resilience (Requirements 46–53):

Requirements ensuring continued mission capability despite loss of personnel, devices, or data flows, addressing the operational continuity dimension of resilience.

The derivation is intentionally scoped: it uses only Block A, and within Block A only the first three of five threat events: writing malicious data, executing malicious programs, and executing valid programs maliciously. These restrictions are not properties inherent to HAT TRICK or to flight software as a problem domain. They define a tractable application of the workflow around a single cybersecurity priority (command integrity), a single flight-software subsystem

(command and data handling (C&DH)), and a focused subset of the HAT TRICK requirement space.

Within that chosen scope, the excluded blocks lie at the boundary of where on-board flight software can act alone. Block B (access vectors—supply-chain compromise, physical access, authorized-subject abuse) is more naturally addressed at integration time, in the ground segment, or through organizational controls; Block C (Type B resilience) targets mission-level continuity rather than flight-software behavior. Similarly, the two excluded Block A events—denying authorized data and obtaining system data—map primarily to the communications layer (link redundancy, CCSDS SDLS anti-replay) and to mission-operations procedures, with on-board data confidentiality orthogonal to the command-integrity focus. A different mission priority or broader HAT TRICK coverage would admit these blocks and events, but that broader coverage requires mission-level evidence beyond this scope.

4.3 Applying Secure-by-Component to Command and Data Handling

The methodology is exercised against a notional spacecraft command and data handling (C&DH) subsystem. The subsystem is a useful first target because it sits at the intersection of ingress (uplinked commands), on-board action (command dispatch and routing), observability (telemetry generation), persistence (event and data logging), and fault-management input (health monitoring)—each of which corresponds to a distinct attack surface. The example is methodological: it demonstrates how a flight-software subsystem can be decomposed, threat-analyzed, and turned into traceable requirements before committing to a single architecture. Chapter 5 uses the resulting requirement categories as inputs to one architectural realization across additional subsystems.

4.3.1 Decomposition of FSW Functions and Components

The first step in the secure-by-component workflow is to decompose the target subsystem into subcomponents that become secure blocks. For C&DH, the decomposition identifies four primary components [15]:

- **Command Reception and Validation (CR)**: Receives uplinked commands from the ground segment, performs syntactic and semantic validation against the command dictionary and the active operational mode, and routes validated commands to target subsystems over the on-board software bus.
- **Telemetry Generation (TG)**: Aggregates readings from on-board sensors and subsystem status registers, assembles them into telemetry packets with timestamps and integrity fields, and presents the packets for downlink.
- **Data Storage and Logging (DS)**: Writes command, event, and telemetry records to on-board non-volatile storage, maintains redundancy and error-correcting encoding, and serves historical queries for operations and post-flight analysis.
- **Health and Status Monitoring (HS)**: Collects operational metrics (power draw, thermal state, subsystem status) from other services, evaluates them against configured thresholds, and emits alerts and recovery requests when thresholds are exceeded.

Applying the same process to other subsystems such as guidance, navigation, and control (GNC), payload management, and power would test the methodology against different threat surfaces and interface patterns [15]. Figure 4.3 summarizes the notional C&DH data flows used as the basis for the attack-surface and requirements analysis.

4.3.2 Attack Surface Analysis

Each component's attack surface is characterized by its inputs, outputs, and dependencies, primarily data connections to and within the spacecraft.

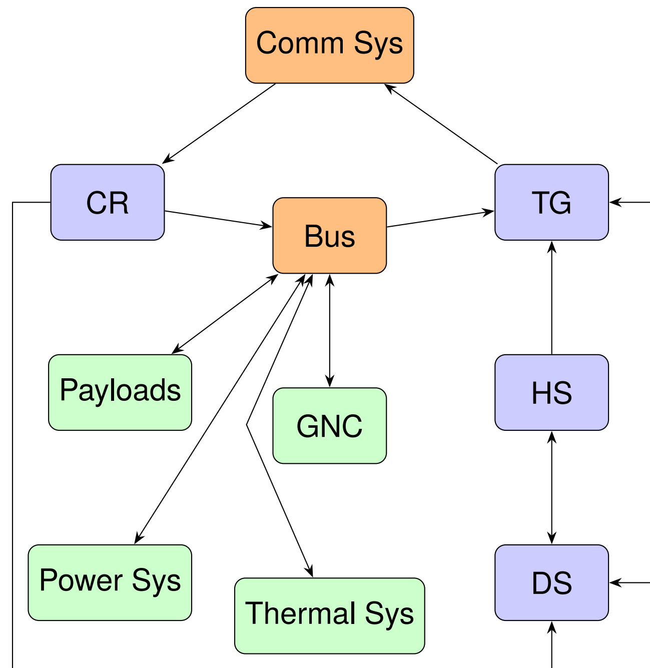


Figure 4.3 C&DH-related data flows. Elements of the notional C&DH decomposition are shown in blue, other spacecraft subsystems are shown in green, and communication or bus elements are shown in orange.

The **CR** component's surface centers on the communication-system input and the configuration interfaces used to load the command dictionary. The lack of message-level authentication identified in Section 3.3.2 means an adversary who reaches the Software Bus can inject commands indistinguishable from legitimate ones.

The **TG** component's surface includes sensor data feeds and bus interfaces; manipulation of either can produce false telemetry. This is the same implicit-trust problem that enables the ground-to-flight command injection chain (Section 3.4.3).

The **DS** component depends on file system, memory management, and data access APIs—the same BSP file system surfaces whose permissive defaults were identified in Section 3.3.3 as persistence mechanisms.

The **HS** component receives subsystem status data and fault logs. Manipulation of these inputs can mask system anomalies or inject false status, directly enabling the fault-management subversion techniques (DE-0001) mapped in Section 3.4.2 [15].

4.3.3 Threat Analysis Using SPARTA and HAT TRICK

For each component, relevant HAT TRICK cyberspace threat events guide the selection and ranking of SPARTA adversarial attack techniques [15]. The command reception and validation component (CDH-CR) illustrates the process. For *Write Malicious Data*, the selected techniques include EX-0003 (Modify Authentication Process) and EX-0010 (Malicious Code)—both target the trust boundary where uplinked commands enter the flight software. The *Execute Malicious Programs* event maps to EX-0004 (Compromise Boot Memory) and DE-0007 (Rootkit), addressing persistence: an adversary who compromises the boot chain or installs a rootkit gains a foothold that survives resets. *Execute Valid Programs Maliciously* maps to EX-0006 (Disable/Bypass Encryption) and EX-0012 (Modify On-Board Values), capturing the scenario where legitimate commands are repurposed—encryption is disabled to expose telemetry, or on-board parameters are altered to degrade system behavior without injecting new code.

The same analysis applied to TG, DS, and HS yields largely overlapping technique sets, because all four components handle mission-critical data and share bus interfaces. Rootkits (DE-0007), malicious code injection (EX-0010), and authentication subversion (EX-0003) appear across multiple components; component-specific techniques such as telemetry downlink manipulation (DE-0003.06) and sensor data falsification (EX-0014.03) reflect each component's unique attack surface. The full per-component technique mappings are tabulated in Curbo and Falco [15].

4.3.4 Identification of Secure-by-Design Principles

The selected SPARTA techniques are now mapped to cyber resilience techniques from the NIST SP 800-160 Vol. 2 catalog that can mitigate them [8]. Of the fourteen resilience techniques in the catalog, five are selected for the command reception component on the basis of fit to the CR threat set: Substantiated Integrity to verify command authenticity before processing; Segmentation to isolate breaches and prevent system-wide impact; Adaptive Response to adjust validation mechanisms in real time; Redundancy to maintain integrity through backup

processes; and Diversity to employ multiple validation methods and avoid single points of failure [15].

The same five techniques recur across TG, DS, and HS, though each component weights them differently based on its threat profile. Telemetry generation emphasizes Diversity (varied timing mechanisms to resist coordinated attacks) and Adaptive Response (dynamic transmission adjustment). Data storage emphasizes Redundancy (backup memory and failover) and Segmentation (process isolation to contain rootkits). Health monitoring combines all five, since the rest of the system depends on it to detect compromise.

This convergence is not coincidental. All four C&DH components handle mission-critical data and share bus interfaces, so a breach in one can cascade into the others. The layering of integrity validation, segmentation, and redundancy across every component reflects this interdependence: no single failure should compromise command integrity, and the software must be able to reconfigure data paths or isolate compromised segments in real time.

4.4 Testable Cyber Requirements Generation

The component-level threat and resilience analysis becomes concrete in verifiable "shall" statements.

4.4.1 Redesign of Components into Secure Blocks

The four components are now redesigned into secure blocks, each encapsulating the original component's functionality with the identified resilience techniques integrated. The secure-by-component approach defines both a minimum and a maximum security design for each secure block. The example here uses the maximum-security design because it best exposes the full requirement set for a high-consequence command-integrity objective; the choice is not a claim that every mission or subsystem must select the maximum design [15].

Two methods are used to fully develop the detailed requirements. First, the pre-defined HAT

TRICK requirements generate a set of high-level “shall” statements for each component, tailored to the specific cyber resilience principles selected. Second, the “shall” statements are further decomposed into detailed requirements that approach the level of specific implementation details.

The full list of standard HAT TRICK requirements [49, Appendix A] can be tailored based on system context and operational constraints. Doing this for the C&DH components results in a list of 15 main requirements covering unauthorized data, unauthorized programs, and unauthorized execution of authorized programs. Using the selected NIST 800-160 Vol. 2 cyber resilience principles, these 15 requirements are further decomposed into detailed requirements addressing the specific security needs of each component [15].

4.4.2 Representative Requirements for Command Reception and Validation

The command reception and validation component illustrates the derivation process. Each representative requirement is traceable to a specific threat (via SPARTA and HAT TRICK), a specific resilience principle (via NIST SP 800-160 Vol. 2), and a specific interface contract within the C&DH architecture. The sub-requirements use flight-software-specific framing rather than general-purpose IT access-control language [15]:

- **CR-1** (Substantiated Integrity): The command reception system shall prevent unauthorized data from being written to the command processing subsystem.
  - **CR-1-1:** The system shall authenticate each received command frame against a verified ground-authority key before the frame enters downstream processing.
  - **CR-1-2:** The system shall validate received commands against CCSDS packet syntax and against the command dictionary’s permitted opcodes, argument types, and argument ranges for the active operational mode.
  - **CR-1-3:** The system shall emit a structured event and increment a monotonic rejection counter for each command that fails authentication, parsing, or dictionary

validation, with source identifier, timestamp, and stable rejection reason.

- **CR-3** (Adaptive Response): Upon detection of unauthorized data, the system shall report the rejection to the responsible telemetry channel in real time, within the bounds of the active mode's downlink schedule.
  - **CR-3-1:** Each rejection event shall include the originating frame identifier (when available), the validation stage that rejected it, and a stable event code from the authoritative event catalog.
  - **CR-3-2:** Rejection events shall be persisted to the on-board event log with monotonic timestamps and a sequence number, and shall survive service restart without loss over the storage window defined by the mission configuration.
  - **CR-3-3:** Events classified as security-relevant shall be distinguishable in the downlink stream from nominal fault events and shall be prioritized above routine telemetry when downlink contention occurs.
- **CR-5** (Segmentation, Substantiated Integrity): The command reception service shall isolate rejected or unverified data from all other on-board services so that no downstream subsystem can act on it.
  - **CR-5-1:** Inter-service communication shall be mediated by a bus that enforces topic-level authorization; only the command service shall be permitted to publish validated commands, and only authorized services shall be permitted to subscribe.
  - **CR-5-2:** Every cross-service message shall carry a source-service identifier and shall be validated against the bus authorization table before delivery; messages failing this check shall be dropped, counted, and reported as structured events.
  - **CR-5-3:** The command service, the bus, and downstream subscribers shall execute in operating-system-enforced isolated address spaces with no shared memory, so that a compromise of one service cannot read or modify the queues of another.

The traceability from Chapter 3 findings to these requirements is explicit. CR-1 addresses the Software Bus's lack of publish-side restrictions (Section 3.3.2) and the absence of command authentication that enables the ground-to-flight injection chain. CR-5 addresses the single-address-space risk that allows a single compromise to become system-wide (Section 3.3.2). CR-3 addresses the observability gap created by the flat execution model, where a compromised component can suppress the telemetry that fault managers depend on (Section 3.4.2, DE-0001).

Alcyone is one architectural realization of these abstract requirements. The important point here is not the eventual service naming, but that the requirements already expose measurable verification criteria: authentication must occur before downstream processing, rejected commands must produce observable evidence, and isolation must be enforced at a boundary that can be tested. These are the criteria by which any later architecture will be judged.

4.4.3 Per-Component Implementation Features

The cyber resilience requirements drive specific implementation feature families for each C&DH component, mapping "shall" statements to concrete technical mechanisms. The command reception component illustrates the pattern most clearly; the full CR-1 through CR-15 catalog appears in the generated requirements artifact and supporting paper treatment [15]:

Requirements CR-1, CR-5, CR-8, and CR-12 (preventing unauthorized data, unauthorized program execution, and malicious code) imply concrete enforcement mechanisms such as cryptographic validation of incoming commands, explicit freshness or execution counters, and containment boundaries that prevent a compromised component from publishing arbitrary commands. CR-4-2, CR-11, and CR-15 (detection and mitigation of unauthorized execution) imply layered verification and monitoring rather than a single check. Adaptive-response requirements across the catalog imply observable response mechanisms such as quarantine, rejection logging, and mode-dependent handling of suspect traffic.

The other three components follow the same requirement-to-mechanism pattern. Telemetry generation requires integrity-preserving data provenance and prioritization rules; data storage

requires integrity checks, bounded persistence, and failure-tolerant storage behavior; health monitoring requires trustworthy status inputs and explicit reporting paths. Different architectures may realize these requirements through different isolation substrates or service decompositions, but the requirement-level obligation remains the same. The full per-component implementation feature mappings are in Curbo and Falco [15].

4.4.4 Requirements Categories and Traceability

The C&DH requirements derived via secure-by-component fall into recurring categories that reflect the mission resilience priority and the NIST SP 800-160 Vol. 2 principles selected [15]:

- **Authority and privilege**: Explicit authorization checks for command execution and configuration changes, driven by Substantiated Integrity and role-based access requirements.
- **Interface integrity**: Authenticated and integrity-checked messages across internal buses and external links, driven by cryptographic validation and data provenance requirements.
- **Isolation**: Containment boundaries between C&DH functions and other subsystems to prevent lateral movement, driven by Segmentation requirements.
- **Monitoring and response**: Observability hooks that support detection, safe-mode transitions, and recovery under attack, driven by Adaptive Response requirements.

The point of categorization is to preserve traceability from threat techniques (Chapter 3) through secure-by-design approaches to concrete enforcement requirements that can be verified in test and integration [15]. Table 4.1 shows how these categories map to the NIST 800-160 Vol. 2 resilience techniques and HAT TRICK requirements that generated them. Independent work on minimum requirements for space systems [14] arrived at converging categories through fault-tree analysis, providing external validation: privilege management, usage restriction, dynamic segmentation, and integrity checks emerge as necessary properties regardless of the derivation method.

Table 4.1 Traceability from the four requirement categories to the NIST 800-160 Vol. 2 resilience techniques and to numbered requirements in HAT TRICK Block A. The HT-n labels in the right column correspond to Block A requirement number n in Adler et al. [49, Appendix A].

Requirement Category	NIST 800-160v2 Technique	HAT TRICK Block A
Authority & privilege	Privilege Restriction	• HT-1 (Prevent unauthorized data) • HT-4 (Neutralize) • HT-12 (Prevent unauthorized execution)
Interface integrity	Substantiated Integrity	• HT-1 (Prevent unauthorized data) • HT-2 (Detect) • HT-3 (Report)
Isolation	Segmentation, Redundancy	• HT-5 (Isolate unauthorized data) • HT-11 (Neutralize unauthorized programs)
Monitoring & response	Adaptive Response, Analytic Monitoring	• HT-6 (Monitor anomalous behavior) • HT-7 (Report) • HT-13–15 (Detect/report/neutralize unauthorized execution)

4.4.5 Verification Approaches and Testability Criteria

A cyber requirement is testable only when it can be verified through observable behavior or reviewable artifacts. The secure-by-component workflow achieves that by writing requirements at interfaces and enforcement points — where data enters a component, where privilege is exercised, and where cross-boundary communication occurs [15]. This aligns naturally with the resilience principles used throughout the chapter: privilege management, segmentation, substantiated integrity, and adaptive response become concrete only when tied to a specific component boundary and a specific interface contract.

Verification strategy is part of the requirement, not an afterthought. For flight software, verification methods range from static checks (build-time configuration assertions, interface conformance tests) to dynamic tests (fault injection, adversarial message injection, and recovery drills). Requirements that cannot be observed or verified in a software-in-the-loop environment are still sometimes useful as design intent, but they do not provide evidence: requirements are

meaningful to the extent that they influence measurable detection, containment, and recovery behavior [15].

4.4.6 Tooling and Evolution

Applying the secure-by-component methodology to one C&DH subsystem produced a tractable shall-statement catalog. Extending the same methodology to the full Alcyone service set — supervisory services, operational support services, mission applications, device drivers, and the platform interface — exposed a different problem: keeping the resulting artifacts linked, current, and machine-readable as the architecture evolved during implementation.

The response was *sextant*, an artifact-management tool developed alongside Alcyone to capture the systems-engineering lifecycle as structured, cross-referenced data rather than as parallel prose documents. Sextant organizes artifact storage around IEEE 15288 (lifecycle), 29148 (concept of operations and requirements), 42010 (architecture views), and 29119 (test process) as its design schema, with project-specific frameworks such as STRIDE, SPARTA, and NIST SP 800-53 configured against that schema rather than hardcoded. Each artifact type lives in a structured TOML store with typed identifiers (`FR-*` for functional requirements, `TA-*` for threat actors, `TC-*` for test cases, and others); cross-references between artifacts are validated against a formal entity-relationship model. The tool surfaces traceability gaps, tracks requirement coverage over time, and emits Markdown reports, JSON traceability views, ReqIF and SysML interoperability exports, and an HTML documentation site. Several tables and figures in this dissertation are sextant-generated.

Table 4.2 lists the artifact types sextant manages for Alcyone with their current populations. The 89-requirement set is one entry in that linked graph; 80 observables, 19 fault definitions, 20 threat-actor identities, and 8 architectural decision records are also captured. The C&DH walk in Section 4.3 above remains the published methodology applied to one subsystem; sextant is what carries that methodology forward across the architecture as services were added, threat coverage broadened from C&DH to the full service map, and architectural decisions were

recorded against their requirement and threat context. Appendix B lists the current top-level requirement set as one view of that linked graph.

Table 4.2 Sextant artifact types managed for the Alcyone systems-engineering lifecycle.

Artifact type	Count	Purpose
Stakeholder records	7	Mission stakeholders and concerns
ConOps scenarios	44	Operational scenarios drawn from the mission profile
Threat-actor identities	20	Adversary classes and assumed capabilities
Faults	19	Fault classes and propagation paths
Hazards	4	Safety-relevant hazards
Requirements	89	Threat-derived shall-statements (34 top-level, 55 sub-requirements)
Architectural views	8	Diagrammatic views of the system
Architectural decisions	8	Decision records with context and consequences
Implementation guidelines	30	Coding and integration guidelines
Observables	80	Measurable evidence points (37 events, 36 counters, 7 gauges)
Test plans	4	Verification and validation campaigns
Code traceability records	68	Code-to-requirement linkage entries
Glossary terms	12	Domain vocabulary

4.5 From Requirements to Architecture Constraints

Some requirements are most naturally met by architectural constraints rather than by point mitigations. Four such constraints follow from the requirement categories and motivate the architecture developed in Chapter 5.

When an interface is implemented in a memory-unsafe language and exposed to attacker-controlled inputs, entire classes of failure become plausible regardless of higher-level policy. The secure-by-component workflow therefore treats memory safety as a resilience-relevant constraint at the lowest levels of the stack rather than as an implementation preference.

Chapter 3 showed that compromise often begins at one interface but becomes mission-impacting through implicit trust and lateral movement. Requirements therefore assume that all inter-component communication is authenticated and validated, with mutual authentication

verifying both sender and receiver before any data exchange [15]. Key management is treated as a requirements-level obligation: keys are provisioned at integration time from an off-board authority; revocation and rotation procedures exist and are exercisable from the ground; and on-board services cannot self-provision or escalate their own keys. Continuous monitoring of communication patterns quarantines suspect channels when anomalies appear.

These requirements also assume an execution environment that can enforce isolation, restrict privilege, and contain compromise. Process isolation, privilege separation, sandboxing, and trusted boot provide this substrate; without them, segmentation requirements reduce to documentation rather than enforcement. Observability is inseparable from enforcement: requirements that specify authentication, integrity checks, and segmentation boundaries must be paired with observability requirements that make enforcement visible, or the system can fail securely in theory while failing silently in practice.

Formal methods provide categorically different evidence than testing: guarantees over all admissible inputs rather than only the cases exercised by tests. Exhaustive verification of large-scale systems remains impractical, so hybrid approaches focus formal methods on the most critical components—command validation, telemetry integrity—while using dynamic analysis elsewhere. Kern's framing of safe coding as type-system-enforced invariants, positioned between ad-hoc testing and full mathematical proof, provides a middle ground that fits the verification budget of flight software [57].

4.6 Summary

A secure-by-component workflow augmented by the HAT TRICK threat framework [15] translated the threat analysis of Chapter 3 into testable flight software requirements. Applied to a representative C&DH subsystem, the process proceeded from component decomposition and attack surface analysis through SPARTA-based threat identification to the selection of NIST SP 800-160 Vol. 2 resilience principles and the generation of traceable "shall" statements at

both the high level and the implementation level. The four recurring requirement categories are authority and privilege, interface integrity, isolation, and monitoring and response. Together they define architectural constraints such as memory safety, zero-trust internal communication, runtime observability, and the integration of formal methods that any later realization must satisfy. Chapter 5 examines one such realization.

Chapter 5
Alcyone: Cyber-Resilient FSW Architecture

Chapter Abstract. Alcyone realizes the requirements of Chapter 4 in a flight software architecture that treats cyber resilience as a design property rather than a post-hoc hardening activity. The architecture organizes command handling, telemetry, supervision, fault management, device control, and mission applications as isolated services connected through a mediated software bus. Rust is used as the implementation language so that boundary and authority assumptions become compile-time enforceable. Service decomposition, authority encoding, formal checks, and SWIL evaluation preserve the same resilience model across host, simulation, and partitioned-runtime paths, linking architecture and assurance into one case.

Chapter Contributions. Four contributions follow. First, it presents the Alcyone architecture: a layered, service-oriented flight software design grounded in linked requirements, threat, architecture, observables, and test artifacts, with each service boundary justified by one or more cyber-resilience goals. Second, it develops the implementation-language rationale, showing why Rust's ownership, type system, and `no_std` support let the architecture preserve the same authority, routing, and containment model across host, SWIL, embedded, and partitioned-runtime paths. Third, it specifies a verification and validation strategy that combines executable reference interoperability for SDLS, bounded model checking (Kani, more than 130 harnesses across 15 crates), deductive proof (Verus, six proof modules), and SWIL evaluation under adversarial conditions. Fourth, it documents the practical lessons from the proof campaign, including two previously hidden implementation defects in a codebase with high test coverage and the proof-model drift findings that motivate co-located proofs and CI integration.

First introduced in prior work [16], Alcyone is the architectural case study that realizes the threat, requirement, and evaluation threads developed in Chapters 3–6. The architecture extends that initial blueprint into a layered service stack grounded in linked requirements, threat,

architecture, and assurance artifacts. The emphasis is not only on the mechanisms themselves, but on whether their boundaries, authority rules, and evidence paths remain legible across implementation and evaluation contexts.

5.1 Related Work

Alcyone's design sits at the intersection of four research areas: reusable flight software frameworks, verified and partitioned embedded systems, memory-safe systems programming, and standards-based spacecraft interfaces. Earlier chapters use these areas to expose the limits of conventional C/C++ hardening and shared-address-space flight stacks. This chapter uses them to justify a flight software architecture whose decomposition carries security authority, not only functional modularity.

Reusable flight software frameworks such as cFS and F' provide mature component models, integration discipline, mission adaptation paths, and operating experience [89, 90]. Their value is strongest where a project needs heritage, an established application ecosystem, and a known ground-integration model. Alcyone does not treat that heritage as a weakness. The difference is the default trust assumption: conventional framework modularity primarily organizes functions for reuse, while Alcyone requires the same decomposition to carry explicit authority, containment, and observable enforcement. That distinction is important because recent work on cFS security continues to show that software-bus and integration defaults can matter as much as individual application code [54].

Formal methods and rigorous analysis are increasingly used around flight software, though often unevenly across the lifecycle. Model checking and theorem proving have been applied to protocol behavior, control logic, and concurrent execution models [91, 92]. Static analysis tools such as Frama-C, Coverity, and Polyspace are more common in practice because they integrate naturally into C-based development workflows, while fuzzing and other dynamic techniques continue to expose parser and boundary-validation faults that static methods may miss [93].

The gap is not lack of tools, but the difficulty of carrying architectural intent, implementation, and evidence forward coherently.

The most mature security-oriented exemplar in embedded systems is the seL4 microkernel, whose functional correctness has been mechanically proved in Isabelle/HOL [55]. The DARPA High-Assurance Cyber Military Systems (HACMS) program showed that building on a verified kernel can eliminate broad classes of exploitable behavior in mission systems [94]. Jero et al. [56] extended this line of work to the space domain with Magnetite, a satellite operating-system approach that uses seL4 as the verified substrate and Rust-based services above it. Those efforts are kernel-centric: the isolation mechanism is the primary assurance target. Alcyone is complementary. It treats architectural authority boundaries, service isolation, and message validation as first-class design objects in the flight software itself, while still treating seL4 as the strongest deployment-oriented isolation substrate for the system (Section 5.4.5).

Memory-safe systems programming supplies the implementation side of the same argument. Rust has become a practical candidate for embedded systems because ownership, borrowing, and `no_std` development can remove common memory-corruption paths without requiring a managed runtime [95]. Government and industry guidance increasingly frames memory safety as a security property for critical software, not merely as a developer-productivity concern [11, 64, 10]. Alcyone applies that reasoning at service boundaries: the language choice is useful because it helps preserve message ownership, typed authority vocabulary, and bounded resource assumptions.

Space standards define another part of the related-work context. CCSDS packet, telecommand, and telemetry transfer-frame standards provide the basic command and telemetry exchange model [96, 97, 98]. CCSDS security and file-transfer standards shape authenticated transfer-frame handling and managed movement of onboard data products [87, 88, 99]. Interface-description standards such as XTCE and SEDS provide metadata expectations for ground-system interoperability and machine-readable interface artifacts [100, 101]. Alcyone uses those standards as external boundary commitments and evidence sources. It does not

allow standards-shaped packet handling to become implicit internal authority, which is the distinction developed in the architecture below.

5.2 System Design Approach

Alcyone begins with threat-informed decomposition. The architecture is specified through linked requirements, threat, architecture, observables, and test artifacts, and each service boundary is justified by one or more cyber-resilience goals: reducing authority overlap, improving containment, making policy enforcement observable, or bounding recovery action. The result is flight software whose security-relevant claims can be tied to explicit mechanisms and later measured.

5.2.1 Core Design Process

The decomposition starts with mission roles, control authority, and attack exposure. Command ingress, telemetry egress, configuration, supervision, device access, file transfer, updates, and mission applications are treated as distinct responsibilities, not conveniences to be bundled into a single image. The key architectural question is not only what the software must do, but which service is allowed to do it, under what conditions, and how the rest of the system observes or constrains that action.

This process follows the secure-by-component logic that independently bounded components expose explicit interfaces, minimal authority, and observable failure behavior [85]. In Alcyone, the unit of decomposition is the isolated service. Each service is assigned a role, an authority scope, a set of permitted message classes, and a recovery relationship with the supervisor. That makes the architecture legible in both nominal and adversarial terms: the same decomposition that supports modular testing also constrains lateral movement and ambiguous control paths.

Requirements are attached to services early, before implementation hardens boundary choices into code. Threat-derived requirements govern uplink validation, bus mediation, recovery

authority, telemetry integrity, bounded queues, configuration integrity, and observability. Each requirement also carries an intended evidence path: test, demonstration, analysis, proof, or some combination. That requirement-to-mechanism-to-evidence chain is what makes the architecture's claims auditable rather than narrative.

5.2.2 Mission Context and Scope

A representative LEO science mission provides the case-study context for the design. The mission profile assumes intermittent ground contact, autonomous operations between contacts, multi-instrument payload activity, and limited ability to patch or manually intervene once deployed. These assumptions are not unique to one mission, but they force concrete architectural choices: command ingress must remain narrow and auditable, telemetry must stay useful under degraded conditions, recovery must be bounded and auditable, and mission applications must not acquire control authority simply because they are operationally important.

That context leads directly to the service set used by Alcyone. The system must authenticate and authorize commands, serialize telemetry onto standards-based downlink paths, monitor service liveness and device behavior, preserve critical state across resets, and allow autonomy without allowing any one service to become a hidden authority hub. The case study is not presented as universal; it is used to keep the architectural choices concrete.

5.2.3 Design Scope and Assumptions

Alcyone is a research flight software system, not a flight-qualified program baseline. It is implemented in Rust and evaluated in host, SWIL, and partitioned-runtime settings, but it does not claim qualification on flight-certified hardware. Host and SWIL paths carry the heaviest empirical exercise; deployment-oriented isolation paths are present to evaluate architectural enforcement, not to certify a complete flight platform.

The design assumes a clean-slate Rust architecture. It does not attempt to harden a mixed-language heritage stack, and the evaluation focuses on software, platform, and architectural

behavior. Radiation effects, certification qualification, and every deep-space mission regime are outside the scope of the case study.

The recovery model also depends on execution substrate. The hosted-Linux path emits a heartbeat consumed by an external sidecar; in-process recovery presumes the flight-software process is alive, so process-kill scenarios measure sidecar latency, not internal mean-time-to-recover. Chapter 6's evaluation runs on seL4, where each service occupies a capability-isolated protection domain with its own scheduling context and process-level loss is not a meaningful failure mode.

5.3 Driving Concept and Formal Requirements

Alcyone is grounded in a linked artifact set: a requirements specification, a threat model, an architecture description, a test document, an observables catalog, and a traceability matrix.

5.3.1 Cyber Resilience Principles

Four principles govern the architecture. First, *OS-enforced isolation*: services are intended to run as isolated OS tasks or protection domains rather than as one monolithic flight image. Second, *explicit authority boundaries*: command, recovery, device access, and configuration authority are centralized and non-delegable. Third, the *secure block model*: every service is a bounded component with explicit ingress and egress paths, policy checks, and observable fault behavior. Fourth, *total observability*: every security-relevant rejection, drop, recovery action, and mode change must be visible through events, telemetry, or counters.

Together, these principles map the NIST resilience functions onto concrete architectural mechanisms. Isolation and explicit authority help the system *withstand* faults or compromise by keeping authority from spreading across service boundaries. Structured detection, recovery, and bounded restart policies provide the *recover* path. Events, telemetry, counters, and enforcement traces support *anticipate* by making baselines, anomalies, and fault evidence measurable.

Mode-dependent policy shifts and reconfiguration provide the basis for *adapt*. Resilience claims can therefore point to specific services and interfaces instead of a general security posture.

Alcyone's assurance posture is therefore *by-construction*: each resilience claim is enforced by a specific mechanism — Rust's type system, kernel-mediated isolation, bus-level access-control tables, or the supervisor's bounded action set. Reviewing a claim starts with the mechanism that enforces it and the evidence it emits.

The principles are intentionally architecture-native, and their relationship to the Chapter 4 requirement drivers is selective, not one-to-one. Substantiated integrity is realized through staged command validation, message provenance checks, configuration validation, and update staging. Segmentation is realized through service decomposition, mediated message routing, independent device-driver endpoints, and task or protection-domain isolation. Privilege restriction is realized by assigning command ingress, configuration, recovery, and device-control authority to specific components so those powers do not become ambient capabilities across the stack. Adaptive response is realized as an evidence-to-action chain: monitoring and event collection produce observations, fault and intrusion logic classify them, and the supervisor applies bounded recovery and mode policy. Redundancy and diversity remain relevant to mission assurance, but they are not foregrounded here unless realized by a specific Alcyone mechanism.

5.3.2 Threat Model

Alcyone's threat model adopts an assume-breach posture. Any component reachable from the CCSDS uplink boundary is treated as a potential adversarial input path, and a compromised onboard service is treated as a plausible internal adversary rather than as an impossible state. Resilience therefore depends less on "perfect prevention" than on detection, containment, and bounded recovery after some boundary has already been crossed.

That posture directly reflects the recurring gaps identified in Chapter 3: permissive internal message fabrics, developer-oriented interfaces that persist into operational builds, and single-

Table 5.1 Alcyone cyber resilience principles and their relationship to Chapter 4 requirement drivers.

Architecture Principle	Requirement Driver	Architectural Realization
OS-enforced isolation	Segmentation	Services are structured for separate tasks or protection domains rather than a single shared-address-space image.
Explicit authority boundaries	Privilege restriction	Command ingress, device control, recovery, and configuration authority are assigned to specific services and not delegated implicitly.
Secure block model	Substantiated integrity	Each service has bounded ingress/egress paths, interface checks, provenance checks, and observable fault behavior rather than undocumented internal trust.
Total observability	Adaptive response	Events, telemetry, counters, and observables are treated as required outputs of enforcement, recovery, and mode adaptation rather than optional diagnostics.

address-space execution models that do not meaningfully contain compromise [13]. In Alcyone, those observations become architectural responses. The command path is narrowed to a single ingress authority. Internal message publication and subscription are constrained by topic access control lists (ACLs). Recovery is centralized in the supervisor rather than coupled to whichever service first notices a fault. Device access is routed through explicit owner-to-driver paths rather than treated as ambient hardware reachability.

The threat model also distinguishes between external and internal attack patterns. External adversaries emphasize uplink compromise, replay, telemetry manipulation, and key-management attacks. Internal adversaries emphasize lateral movement across the software bus, unauthorized device commanding, and attempts to interfere with detection or recovery. That distinction is why the architecture couples command validation, bus mediation, supervision, fault management, and behavioral intrusion detection rather than treating them as separate concerns.

5.3.3 Requirements and Traceability Basis

The formal design basis for Alcyone is captured in generated requirements, test, and traceability artifacts. The current requirements specification is organized around 34 top-level requirements — 27 functional (FR) and 7 interface (IR) — with each top-level requirement decomposed into sub-requirements that name concrete enforcement obligations (89 individual shall-statements in total). The set spans system behavior, cyber-resilience constraints, interface definitions, performance, and observability; B lists the top-level entries.

The top-level set is one application of the Chapter 4 workflow, not a universal minimum set. Section 4.3 walks through the secure-by-component derivation for C&DH; the Alcyone requirement set extends the same threat-to-category-to-mechanism logic to supervision, support, device-handling, and platform-interface services. Engineering judgment from the Chapter 3 attack-surface findings supplements that derivation where the service map exposes risks beyond the C&DH example.

The Chapter 4 C&DH walk is therefore methodological: it shows how a secure-by-component derivation is performed. The current Alcyone requirement set documents the same derivation reshaped for the full service map and tooled in the artifact-management system described in Section 4.4.6. The test documentation traces those requirements into functional verification, adversarial validation, SWIL plans based on NASA's Operational Simulator for Small Satellites (NOS3), and quantitative resilience assessment plans. The traceability matrix then links stakeholder concerns and threats forward into specifications, code modules, and test cases.

Table 5.2 summarizes how the main requirement drivers appear in the architecture itself.

5.4 High-Level Architecture

Alcyone's high-level architecture follows from the authority boundaries established above. The service layer holds operational authorities and mission functions. A common core supplies message schemas, routing semantics, event handling, and timing interfaces. Platform traits

Table 5.2 Representative traceability from requirement drivers to architectural consequences in Alcyone.

Requirement Driver	Architectural Consequence	Representative Services / Mechanisms
Threat-derived boundary validation	Reject malformed, replayed, or unauthorized inputs at the earliest viable boundary	Staged command parsing, authentication, authorization, and routed-message policy checks
Isolation and least privilege	Separate service roles and make sensitive authority non-delegable	Mediated routing; sole external command-ingress authority; supervised recovery authority; owner-to-device command routing
Observability and measurement	Turn enforcement activity into structured evidence for both operators and experiments	Event aggregation, telemetry downlink, observables catalog, and quantitative resilience metrics
Bounded recovery and degraded operation	Keep recovery decisions explicit, auditable, and mode-aware	Health reports, fault directives, intrusion assessments, supervised restart, and mode policy
Configuration and update integrity	Treat parameter changes, file transfer, and updates as managed control paths rather than ambient capability	Configuration validation, managed file transfer, update staging, and update verification

isolate runtime-specific behavior, while concrete execution substrates provide threads, tasks, or protection domains. This layering lets mission-dependent services vary without changing the authority model that the rest of the chapter evaluates.

At the service layer, Alcyone runs as a set of long-lived services, not as one monolithic flight loop. Each service owns one operational responsibility, subscribes only to specific message classes, publishes only to authorized topics, and has a defined recovery relationship with the supervisor. Most services are restartable. The message-routing service and the supervisor are exceptions because they anchor communication and recovery control. Three boundary classes carry the architecture: service-level authority, standards-shaped external interfaces, and runtime substrates that support and enforce those boundaries.

Figure 5.1 summarizes the layered view.

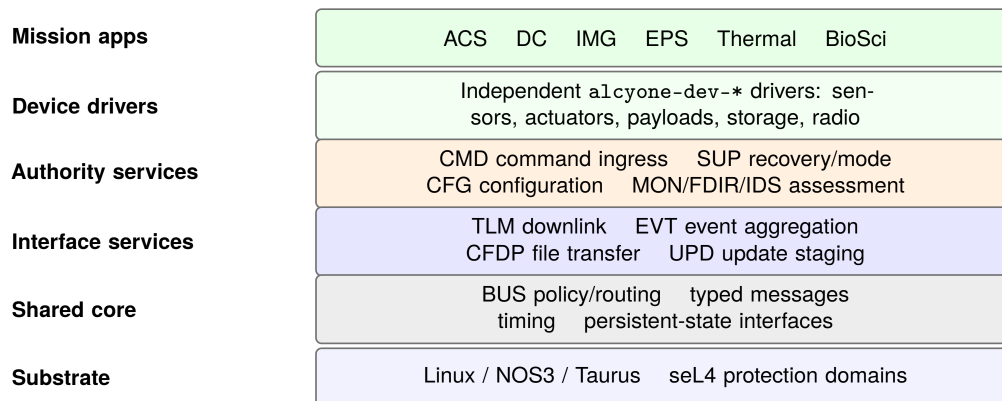


Figure 5.1 Compact layered view of the Alcyone architecture. Mission applications and independent device drivers sit above authority and interface services; the shared BUS, typed-message, timing, and persistence core mediates communication across Linux/NOS3/Taurus and seL4 execution substrates. The figure groups components by architectural role rather than showing one fixed mission build.

Each service group owns a different authority boundary and produces a different class of operational evidence. The decomposition is deliberately wider than a single command-and-data-handling loop and narrower than a generic plug-in framework: each service represents an explicit authority boundary, not just a packaging convenience.

Command and data handling owns external command ingress, telemetry egress, mediated bus routing, and the shared time base. Its evidence includes authentication and replay rejections,

authorization failures, route drops, event counters, and downlink status. Supervision and detection own recovery authority, mode authority, health assessment, fault classification, and intrusion assessment; their evidence appears as health reports, fault directives, intrusion assessments, restart records, and mode transitions.

Operational support owns configuration changes, schedule execution, managed file movement, update staging, and persistent state. Mission and application services own mission behavior under mediated command, device access, and supervised recovery. Device-driver groupings own the hardware-facing source and sink boundaries used by the Chapter 6 runtime. Together, these groups define who may exercise authority and what evidence remains after that authority is used.

5.4.1 Command and Data Handling

Command and data handling defines the spacecraft's primary external boundary, and Alcyone separates that external ingress and egress from internal mediation and evidence collection. The command service (CMD) is the sole external telecommand ingress point: it receives ground-originated commands and turns them into internal actions only after validation succeeds. The telemetry service (TLM) is the sole CCSDS downlink path for housekeeping and mission telemetry. The event service (EVT) collects structured events, counters, and other internal evidence. The software bus service (BUS) carries the mediated traffic that connects the rest of the stack. Keeping these roles distinct prevents the interface boundary, the evidence path, and the message fabric from collapsing into the same component.

The command ingress path is organized as a staged validation pipeline. Structural validation checks the received packet and its framing. Authentication and anti-replay logic then establish whether the command originates from an authorized source. Authorization is the final step: command permissibility is checked against mode and authority policy before the command becomes behavior. This sequence matters architecturally because it separates syntax, identity, and permission instead of collapsing them into one permissive dispatch step.

The first enforcement claim is replay resistance at the authenticated command boundary. The receiver-side invariant for a sliding anti-replay window is that newly advanced sequence numbers may move the window forward, recent out-of-order sequence numbers may still be accepted once, and duplicates or sequence numbers that have fallen behind the retained window are rejected. Alcyone uses the same ordering principle at the command-authentication layer through per-principal monotonic counters, while the SDLS-EP frame path uses the window form formalized in Definition 5.1.

Definition 5.1 (Anti-replay invariant)**.** Let $W = (e, S, w)$ denote the SDLS-EP anti-replay state held by the receiver, where e is the next expected sequence number, S is the set of authenticated sequence numbers already accepted within the retained trailing window, and w is the window span. For an incoming authenticated frame with sequence number n, the receiver accepts the frame if and only if n is at or beyond e, or n is still inside the trailing window and has not already been seen. Omitting wrap-around notation for clarity, $\textsc{Accept}(n, W) \iff n \geq e \vee (e - w \leq n < e \wedge n \notin S)$. On acceptance, the receiver records n, advances e to at least $n + 1$, and prunes S to the retained window.

Algorithm 1 states the operational form of the invariant in pseudocode. The check is $O(1)$ when S is implemented as a bounded bitmap, as in Alcyone's fixed-width replay window, and $O(\log w)$ when represented by a balanced set.

The remaining command checks are better understood as ordered gates than as a novel algorithm. A command that fails structural validation never reaches authentication. A command that fails authentication or replay checking never reaches authorization. A command that authenticates but lacks authority under the active mode, command ACL, or dual-authorization policy is rejected before dispatch. The ordering is the architectural property: each stage narrows the set of commands that can affect internal state, and no later stage can recover authority that an earlier stage denied.

The telemetry service (TLM) carries aggregated observability data and mission telemetry to the downlink while preserving the distinction between internal evidence and external transport.

Algorithm 1 Anti-replay window check for an incoming SDLS-EP frame.

Require: sequence number n; window state $W = (e, S, w)$
Ensure: ACCEPT or REJECT

1: **if** $n \geq e$ **then**
2: $S \leftarrow (S \cap [n - w + 1, n - 1]) \cup \{n\}$
3: $e \leftarrow n + 1$
4: **return** ACCEPT
5: **end if**
6: **if** $n < e - w$ **then**
7: **return** REJECT
8: **end if**
9: **if** $n \in S$ **then**
10: **return** REJECT ▷ replay
11: **end if**
12: $S \leftarrow S \cup \{n\}$
13: **return** ACCEPT

EVT serves as the observability hub: it receives events from services, applies filtering and throttling policy, preserves counters needed for measurement, and feeds downstream consumers such as supervision, detection, and telemetry. Observability is therefore part of the architecture, not an auxiliary logging feature.

BUS is where internal reachability becomes a policy decision, not an emergent property of the message fabric. A service may publish or subscribe only on topics admitted by the active ACL, and the sender identity attached to a message comes from the runtime endpoint and bus binding, not from a field the service can choose. Messages are copied between services instead of shared through mutable buffers, queue depths are bounded, and backpressure is reported explicitly. Internal traffic is therefore observable and constrainable even after it has crossed the external command boundary.

The enforcement path through BUS is intentionally narrow. When a frame arrives, the transport has already bound it to a sender identity. BUS verifies the SecOC MAC with that sender's derived key, sets the source field to the verified identity, evaluates the topic-ACL table, and routes only through the static topology. Each outbound copy is signed under the destination's derived key. SUP-originated mode-control frames are handled before ordinary

routing so the supervisor can switch the active inter-protection-data-unit group and narrow reachability between nominal and safe operation without restarting services. Six counters make those outcomes visible: frames received, frames routed, and frames dropped because no route exists, authentication failed, the ACL denied routing, or mode control was invalid.

Replay rejection at the authenticated command boundary (Definition 5.1) applies to external commands and is in force on every deployed substrate; it is independent of the internal-bus path. Internal-bus traffic is authenticated and identity-stamped on the inbound path, but inbound freshness enforcement is staged rather than wired: a freshness tracker is implemented and not yet integrated into the router. Internal traffic is therefore gated by identity, ACL, and routing-policy checks rather than treated as trustworthy because it lives inside the flight image.

Authority is not the same thing as arbitrary reachability on the bus. A route represents authority only when the routing policy, command policy, and current operating mode all admit the sender, receiver, and action.

Definition 5.2 (Service authority). Let $\tau(a)$ be the message type whose accepted delivery can request action a, and let P be the active routing and command policy. A service s has *authority* over a in operating context C only if P admits s as a publisher for $\tau(a)$, the owning component is an admitted receiver, and the current mode and Supervisor policy in C permit a from s. A service without authority over a has no valid bus-mediated path for causing a to occur.

The routing table is the implementation artifact that preserves that authority relation. For each message type, its admitted publishers and subscribers must be registered bus entities allowed by policy; when a topic carries command-bearing messages, its permitted publishers must be services with authority over the associated action. Some topics legitimately have more than one publisher, such as ground-originated commands and stored-command traffic, so singleton publication is a policy property for non-delegable actions rather than a universal rule for every message class.

5.4.2 Supervision and Recovery

Alcyone separates health observation, fault classification, cyber detection, and recovery authority into distinct services. The design goal is bounded control: the service that notices a problem is not automatically the service that can reconfigure the spacecraft in response.

The monitor service (MON) provides the first layer of operational evidence. It periodically checks service liveness through heartbeat-style health reporting, tracks degraded or missing responses against configured thresholds, aggregates health state, and reports that state to the supervisor. MON can therefore tell the rest of the architecture that something is wrong, but it does not restart services or change spacecraft mode. That boundary keeps monitoring logic observable and testable without turning it into a hidden recovery authority.

The fault detection, isolation, and recovery (FDIR) service and the intrusion detection service (IDS) provide two different assessment paths. FDIR is the rule-based fault analysis service: it correlates reported faults against threshold and timing logic and produces classified recovery recommendations. IDS is the behavioral analysis service: it evaluates bus traffic patterns, command rates, event streams, and cross-service anomalies to identify probes, floods, anomalous baselines, and other cyber-relevant patterns that do not always present as one direct fault. EVT and MON supply much of the evidence that makes those assessments possible, but both services remain advisory with respect to system-wide action.

That split is deliberate. Fault and intrusion logic work from different evidence and tolerate different false-positive costs, but their recovery options draw from the same limited set of restarts, degradations, and safe-mode transitions. If each detector held direct recovery authority, the architecture would gain competing action paths that could conflict or exhaust bounded recovery budgets. Alcyone therefore lets FDIR and IDS share recovery infrastructure without sharing recovery authority: both can classify and recommend, but only SUP can reconfigure the spacecraft.

The supervisor service (SUP) is the sole recovery and mode authority. It receives health reports, fault classifications, and threat assessments; applies bounded restart and escalation

policies; and maintains the system mode state machine. It is therefore not just another management service, but the trust anchor that decides whether the correct response is restart, degradation, safe mode, or continued operation. Operationally, SUP manages six system modes: Startup, Nominal, Degraded, Safe, Recovery, and CommLoss. The key invariant is authority monotonicity. Fault- or anomaly-triggered transitions do not widen command authority; they maintain or reduce it. That property matters as much for cyber resilience as for safety because it prevents the recovery path from becoming an unguarded privilege-escalation path.

For each response action class, SUP maintains a bounded retry budget over a configured operating-time window. It may issue a response only while that action class remains within budget; once the budget is exhausted, SUP escalates rather than retrying indefinitely.

This budget rule is the mechanism that makes detector compromise survivable. MON, FDIR, and IDS can produce evidence and recommendations, but they cannot directly restart services, change modes, or widen command authority. SUP first maps a recommendation into the closed response set, rejects responses that are not admitted by the current mode and policy, and then applies the relevant budget before issuing any action. The result is not merely rate limiting: it preserves a single recovery authority, prevents competing detectors from consuming unbounded recovery attempts, and keeps anomaly-triggered transitions monotonic with respect to command authority.

5.4.3 Support Services and Mission/Application Layer

Support services and mission applications provide persistence, autonomy, file movement, device interaction, and mission-specific behavior. These services and drivers turn a secure command path into a usable spacecraft software stack while keeping high-leverage operational functions explicit instead of embedding them as hidden helper logic. Configuration, sequencing, file transfer, software update, device control, and mission applications are therefore separated so that their policy boundaries can be stated, enforced, and observed.

Five support services carry most of that operational weight. The configuration service (CFG)

owns configuration storage, validation, and controlled parameter distribution. The scheduler service (SCH) provides time-tagged or stored-command execution for autonomous operations without bypassing the command or supervisor model. The file-transfer service (CFDP) supplies managed ground-to-flight and flight-to-ground transfer behavior. The update service (UPD) stages and validates software-update flows before they can affect the operational image. The storage service subscribes to the same downlink stream as TLM but retains packets in a bounded drop-head ring for ground-commanded replay. Observability therefore survives degraded contact, and replay traffic does not contend with the live downlink.

These services sit outside the primary command path, but they remain critical resilience boundaries because configuration, sequencing, file transfer, and software update are all high-leverage paths for either recovery or compromise.

Mission and application logic is likewise separated from core control authority. In the current Alcyone architecture, there is no single device-manager service acting as the exclusive hardware gate. Instead, device drivers are independent bus participants with explicit identities, shared driver infrastructure, and owner-to-device routing policy. The attitude control service (ACS) owns attitude-control devices and forwards bounded actuation requests to the relevant reaction-wheel, magnetic-torquer, sensor, or thruster drivers. The data collection service (DC) owns storage-oriented device flows such as recorder and memory-device interactions.

Spacecraft support services such as the electrical-power service (EPS) and thermal management can be enabled alongside mission-specific services such as the imaging service (IMG) and other science-payload functions, but they remain application-layer services rather than alternate control planes. Operational importance does not grant command or recovery authority.

Across the full service architecture, a small set of design patterns remains stable. Ground links terminate at the command and telemetry services. Internal traffic flows through BUS policy. Device-facing operations use explicit owner-to-driver command routes rather than ambient hardware access. Persistent state, parameter changes, file transfer, and updates move through CFG, CFDP, and UPD rather than through ambient filesystem or debug interfaces. These stable

boundaries provide the basis for the comparison with cFS and F' in Section 5.7.

5.4.4 Standards-Shaped External Interfaces

Alcyone's external interfaces are standards-informed rather than proprietary. The architecture uses CCSDS Space Packet Protocol framing as the primary packet boundary, with command and telemetry packets terminated at CMD and TLM rather than exposed as the native internal bus format [96]. Packet Utilization Standard (PUS) service and subtype conventions shape command and telemetry semantics: request verification, housekeeping, event reporting, function management, scheduling, parameter management, and file-management traffic are expressed through PUS-style operation categories while internal service routing remains an Alcyone concern [102]. This separation is deliberate. Ground systems see familiar command and telemetry semantics, while the flight software preserves typed internal messages and authority checks instead of allowing external packet structure to become ambient authority inside the vehicle.

At the data-link layer, Alcyone supports CCSDS telecommand and telemetry transfer-frame profiles for uplink and downlink framing [97, 98]. SDLS provides the security model for authenticated and confidential transfer-frame handling where the deployment enables that path [87, 88]. Unified Space Data Link Protocol (USLP) can occupy the same link-layer role for deployments that select it, but the architecture does not depend on USLP specifically [103]. Link-layer framing and security terminate at the external boundary services; they do not replace service-level authorization, bus policy, or supervisor-controlled recovery.

Other standards appear as narrower interface commitments. CFDP provides the file-transfer path for managed movement of data products, configuration artifacts, and update payloads [99]. Ground-system interoperability uses machine-readable interface descriptions, but the operational source of truth lives in the Rust type system: telemetry, telecommand, and enum derives (`TlmPayload`, `TcPayload`, and `XtceEnum`) attach wire formats and ground-system metadata directly to the type definitions. `BusPayload` applies the same type-derived serialization

discipline to internal bus messages, but those messages are not ground-visible and do not carry XTCE metadata. XTCE dictionaries plus SEDS package files are generated from or validated against the ground-visible types rather than authored independently [100, 101]. Standards conformance is an architectural boundary and an evidence source. Where a standard defines wire behavior, the implementation is cross-checked against it; where a security requirement conflicts with a permissive standards mechanism, the divergence is explicit and justified.

5.4.5 Runtime and Hardware Abstraction

Runtime variation is itself an authority boundary in Alcyone. Service logic depends on a small set of platform traits instead of direct Linux, NOS3, or seL4 APIs. Services are generic over five recurring interfaces: `Bus`, `Clock`, `PersistentStore`, `PlatformControl`, and `Transport`. Platform-specific behavior enters through a bounded trait surface, so portability depends on implementing the runtime boundary rather than rewriting flight logic.

Each trait captures one class of platform dependency. `Bus` abstracts mediated inter-service messaging. `Clock` provides the time base used for scheduling, timeouts, and time stamping. `PersistentStore` encapsulates retained state such as configuration and durable logs. `PlatformControl` exposes lifecycle operations such as restart or shutdown only to services that genuinely need them. `Transport` covers runtime-specific external I/O paths, such as uplink, downlink, or other off-board communication channels. Taken together, these traits let services preserve one architectural model while moving across host, SWIL, and partitioned-runtime environments.

That trait-based split supports two execution classes. Linux is the host environment for service-logic build, unit testing, and early integration. seL4 under QEMU emulation is the deployment-oriented isolation substrate and the runtime under which Chapter 6's adversarial campaigns exercise Alcyone. The same service logic moves across both substrates while spatial and temporal isolation are localized to the kernel on seL4.

Service authority depends on temporal as well as spatial isolation. On seL4, Alcyone maps

the service decomposition onto protection domains: BUS is the mediated message hub, the supervisory and command/telemetry/event services occupy separate domains, mission and persistence services follow, and four device-simulation domains feed the evaluation harness. The mapping is direct rather than incidental — domains correspond to architectural roles — and the substrate reinforces Alcyone's existing authority model in two ways. Capability-based access control prevents a service from naming any resource it has not been granted, regardless of implementation defects. MCS scheduling-context capabilities bind a budget and period to each domain, so CPU overrun becomes a bounded scheduling event rather than silent degradation of the control plane [104]. The kernel itself carries mechanical functional-correctness proofs at a code size compatible with flight resource budgets, placing it outside the application-layer trust analysis [55, 56].

5.5 Benefits of Rust for Flight Software

Rust's contribution to Alcyone is architectural rather than cosmetic. Ownership and borrowing reinforce message ownership; the type system makes authority vocabulary and resource bounds harder to bypass accidentally; and `no_std`-compatible abstractions keep those patterns viable on embedded targets. Together these properties preserve Alcyone's authority boundaries, state ownership, and bounded-message model across implementation targets. They do not eliminate certification, toolchain, or ecosystem constraints, but they keep the implementation aligned with the architecture's containment model rather than at odds with it.

5.5.1 Safety Guarantees

Rust's ownership and borrowing model removes broad classes of memory-safety defects from the service layer without requiring garbage collection or pervasive runtime checking. For Alcyone, that matters less as a generic software-quality improvement than as a constraint on architectural failure modes. Services exchange owned messages rather than borrowed

mutable aliases into shared state, and safe Rust prevents the out-of-bounds writes, dangling references, and null-pointer behavior that would otherwise undermine the message-copy and bounded-ownership model.

Those guarantees matter for cyber resilience: memory corruption is one of the standard routes by which an input-validation bug becomes an execution-control bug. In Alcyone, the command path, message path, and much of the service logic are written in safe Rust, so the architecture is not relying exclusively on tests or coding standards to preserve containment. The U.S. Office of the National Cyber Director's 2024 call for memory-safe languages in security-critical software [11] converges with earlier CISA and NSA documents that identify memory-safe languages as a structural way to reduce exploitable defect classes [64, 10].

Evidence from large production ecosystems points in the same direction. Shifting security-critical components to memory-safe languages has materially reduced memory-corruption vulnerabilities and narrowed the residual defect surface in deployed systems [105, 61]. The Alcyone-specific claim is narrower: Rust's containment properties show up inside the architecture's service boundaries, where memory-safety bugs would otherwise let a compromised input path reach unrelated state.

Safe Rust also narrows the trusted surface of the implementation. Unsafe code is not treated as forbidden in principle, but it is confined to platform, FFI, and runtime edges rather than distributed across service and mission logic. That concentration matters because it makes the unsafe boundary explicit and reviewable, which is consistent with the architecture's broader preference for explicit rather than ambient authority.

The choice of Rust addresses a specific subset of the safety properties relevant to flight software: memory safety, data-race freedom, and type-level invariants. Other safety dimensions—logical contracts, bounded-time execution, and functional correctness proofs—are addressed elsewhere in the architecture rather than in the language proper. Logical contracts are split between the type system, where authority vocabulary, message classes, and resource bounds are encoded as compile-time constraints, and the formal-methods workflow described in Sec-

tion 5.6, where Kani and Verus discharge bounded and deductive proof obligations respectively. Real-time bounded execution is supplied at coarser granularity by the seL4 substrate rather than by language-level scheduling primitives. Functional correctness proofs are scoped to selected architectural invariants—routing, authorization, anti-replay, protocol round-trips—rather than attempted across the entire system. The aggregate safety claim therefore depends on the composition of these layers, not on Rust alone.

5.5.2 Type System and Interface Contracts

Rust's type system lets Alcyone encode architectural intent into interfaces instead of leaving that intent in comments or integration folklore. Traits define what a service can depend on. Enumerated service identifiers and message classes keep authority vocabulary finite and machine-checkable. Bounded collections and fixed-size buffers keep resource assumptions close to the code that depends on them. The cumulative effect is that many category errors become type errors rather than latent integration defects.

This matters most at protocol and routing boundaries. Packet fields with constrained widths, message classes with fixed semantics, and authority scopes with typed set operations are easier to preserve when they are reflected in the type system instead of reconstructed repeatedly from raw integers and ad hoc checks. For a cyber-resilient architecture, that means fewer opportunities for the implementation to quietly violate its own model. Field-level validation happens once at the parsing or ingress boundary; inward-facing code then carries values whose constraints are encoded in their types rather than re-checked at each use. A packet whose length field is constrained to a known range becomes a typed length once parsed; downstream services consume that typed value without re-validating the bound. This pattern preserves the certification-friendly property described in Section 5.6—runtime invariants are not asserted defensively at every call site—while keeping the trust boundary at a single, reviewable point in the parser.

The same type-level discipline appears in Alcyone's platform boundary. Services are generic

over the platform traits they are allowed to use: for example, SUP depends on `PlatformControl`, while monitoring and command-path services do not. A service that lacks that trait cannot call restart or memory-protection operations without changing its type signature, which turns an authority expansion into an explicit architectural change. The platform traits also use a shared `Sealed` marker convention so that production implementations remain concentrated in platform and test-support crates rather than appearing as casual service-local adapters.

Alcyone also uses derive macros to keep typed interfaces aligned with wire formats and ground-system metadata. Telemetry, command, bus-message, and XTCE enum payloads derive serialization and metadata from the Rust type definition, including field widths, discriminators, and packet metadata. That mechanism does not prove semantic correctness, but it reduces drift between the implementation, the on-wire representation, and the machine-readable interface descriptions used by test and ground tooling.

The crate-per-block organization reinforces the same boundary logic at the build-system level. Services, libraries, device drivers, and platform layers are separated into distinct crates so that dependency structure becomes an architectural artifact rather than a by-product. Rust is not the only language that could support this organization, but it makes the interface boundary more rigid once established.

5.5.3 Performance, Portability, and Embedded Viability

Rust's abstractions remain practical in constrained environments because the language has a compilation profile (`no_std`) that excludes the standard library, the heap allocator, threading primitives, and any implicit OS dependencies. On the seL4 substrate, those omissions are not a limitation but the point: services run with explicit allocators, fixed-capacity collections, and platform-trait-mediated I/O, so memory use and runtime dependencies are visible at the type level rather than smuggled in through standard-library calls. The same flight crates compile on Linux for development by enabling a `std` feature gate where useful; the no-feature default is the flight build. Zero-cost abstractions and bounded collections then let the same architecture span

host-based evaluation, SWIL, and seL4 targets without rewriting the service model around each platform's tooling idiosyncrasies [95].

The bounded-collection choice is concrete rather than stylistic. Alcyone uses fixed payload arrays and `heapless` queues, vectors, strings, and `Deque` instances in flight-facing crates so that queue depth, payload size, and temporary storage are visible resource parameters. Capacity pressure therefore becomes an explicit design condition instead of an accidental heap-growth behavior.

That portability lets the same architecture serve development and deployment-oriented isolation paths without re-authoring the service logic. Linux is the development host. NOS3 and Taurus provide the SWIL evaluation harness, with Alcyone running on seL4 under QEMU emulation as the evaluation runtime. The platform layer absorbs those substrate differences while preserving the service model.

5.5.4 Adoption Constraints

Rust does not remove the harder engineering questions around qualification, supply-chain control, long-term toolchain support, and workforce familiarity. One specific dimension is the form of language specification available for certification: Ada has a prescriptive ISO standard against which multiple compliant implementations exist, while Rust's specification work, including the Ferrocene Language Specification [106], currently documents the behavior of the reference implementation rather than prescribing the language independently of it. For programs whose certification authority requires an implementation-independent specification, this is a real gap that the qualified-toolchain effort alone does not close. Compiler and library maturity are improving quickly, and qualified toolchains such as Ferrocene are materially important, but the certification ecosystem remains younger than the Ada/SPARK and heritage C toolchains common in aerospace [66].

The embedded ecosystem also remains uneven. Some targets and peripheral libraries are well supported, while others still require custom work at the platform boundary. In Alcyone,

that unevenness is manageable because the architecture already isolates platform-specific code, but it remains a real adoption constraint for programs that cannot afford custom runtime integration.

SPARK occupies a different point in the same assurance trade space. Its prescriptive ISO specification, decades of DO-178C qualification heritage, contract-based verification at the language level, and Ada/Ravenscar's first-class concurrency profile fit a flight-software project whose certification authority requires implementation-independent semantics, contract-style proof obligations as a language feature, and a workforce already trained in the toolchain. For a clean-slate research architecture targeting an seL4 deployment substrate, where assurance is distributed across language safety, kernel-mediated isolation, and external proof tools (Kani, Verus, interoperability harnesses), Rust's compositional stack and open ecosystem produce a viable implementation path even where the specification work is still maturing. The two languages address different assurance topologies rather than competing for the same engineering envelope.

Rust suits this architecture because ownership, typing, and `no_std` support preserve the same authority, routing, and containment model across the host, SWIL, and partitioned-runtime paths. Certification ecosystem maturity, supply-chain control, and toolchain qualification remain separate engineering obligations the language alone does not resolve.

5.6 Verification and Validation Strategy

Alcyone's assurance strategy follows the same boundary-oriented logic as the architecture itself. Standards-critical interfaces are checked through executable references and interoperability tests. Implementation invariants at authority, routing, parser, and protocol boundaries are targeted with formal methods. Runtime hygiene checks constrain the unsafe surface, and SWIL evaluation exercises the full system through the same external interfaces it exposes operationally. The result is an evidence stack that spans protocol conformance, code-level invariants, unsafe-boundary discipline, and system behavior under adversarial conditions.

5.6.1 Implementation-Level Assurance

Textual conformance to space standards is often insufficient. For CCSDS security and packet-processing work, Alcyone relies on executable reference behavior where possible. The clearest example is SDLS, which protects the most adversary-exposed path in the architecture: the telecommand uplink. Rather than relying only on local unit tests derived from a reading of the standard, the Rust implementation is validated against NASA CryptoLib through generated test vectors and interoperability testing.

This approach matters because protocol ambiguity often survives careful reading. Two implementations can each appear reasonable against the standard text while still disagreeing on byte ordering, authenticated data construction, or state transitions. Executable cross-validation surfaces those mismatches much earlier than end-stage integration does. In that sense, reference interoperability acts as a form of specification refinement, not merely as a regression test.

It also sharpens the threat model. Adversarial review of the SDLS path exposed issues such as counter-reset handling that were not obvious from nominal implementation testing alone. That experience reinforced a broader design lesson already visible in Chapter 3: threat-modeling and standards interpretation proceed together rather than sequentially.

The same implementation-level assurance story extends beyond standards validation into formal verification and implementation hygiene. Alcyone uses two formal tools for different proof styles. Kani is a bounded model checker for Rust: it executes harnesses over finite state spaces and is well suited to parser logic, routing tables, authorization checks, and other edge-condition-heavy code. Verus is a deductive verification tool: it proves selected invariants over modeled program logic and is better suited to stable structural properties such as authority relations, replay-counter behavior, and protocol invariants.

That division of labor is important to the assurance claim. Because safe Rust eliminates whole classes of undefined behavior at compile time—use-after-free, mutable aliasing, data races, and unchecked pointer arithmetic—the verification conditions that Kani and Verus generate are

framed in terms of program logic rather than language-level semantic hazards. The verifier reasons about whether a routing table preserves its invariants, not whether dereferencing a freed pointer is reachable. That reduction in the residual state space makes bounded model checking tractable for the architectural invariants Alcyone targets.

A secondary consequence is that statically proven invariants do not introduce runtime branches that subsequent coverage analysis must discharge. At the highest aviation-assurance levels, defensive runtime checks for properties that hold by construction can become a verification liability: they create decision points that structural-coverage criteria must cover, and those decisions often cannot be exercised because the failing inputs are unreachable by construction. Encoding the same property in the type system or proving it with Kani avoids that liability without weakening the guarantee, because the property is established once at compile or proof time rather than asserted on every execution.

Kani is the most deeply integrated formal tool in the Alcyone workflow, with more than 130 harnesses across 15 crates. Those harnesses target the residual properties that remain after Rust's safe subset has already removed many memory-safety concerns: enum round-trips, routing-table bounds, authorization soundness, anti-replay behavior, parser and serializer invariants, state-machine correctness, and panic freedom in boundary logic.

Kani is well suited to the bounded state spaces where Alcyone's architectural claims can fail: command authorization, routing, crypto state, fault detection, isolation, and recovery (FDIR), and shared type definitions. It exhaustively explores edge conditions in those crates, which is more useful here than attempting one monolithic proof over the entire flight stack.

The tool has also been productive in practice. The verification guide records the two previously hidden implementation defects surfaced during the proof campaign, along with related hardening findings such as security-relevant constant mismatches and unreachable enum/fault-code paths. The harnesses are also co-located with the code they check and run in the normal development workflow, which reduces the distance between architectural intent and proof maintenance.

Verus complements Kani by handling properties that are awkward to express or insufficiently covered in bounded search. The proof set spans six proof modules covering routing-table invariants, key-store authority relations, replay-counter behavior, telecommand-frame roundtrips, and CCSDS validation logic. These are the kinds of invariants that are both security-relevant and structurally stable enough to justify proof effort.

The main caution is model drift. Verus proofs reason over models of production logic, not the compiled production code directly. That distinction does not invalidate the proofs, but it does mean a "passing proof" is only trustworthy when the model remains meaningfully aligned with the implementation. For that reason, Verus is presented as part of the evidence base for selected architectural invariants rather than as blanket proof coverage for the full system. This caveat is not unique to Alcyone. Recent systematizations of cyber-physical systems resilience explicitly identify the semantic gap between verified models and deployed firmware binaries as one of the persistent structural gaps in formal-methods practice [107]. Co-locating Verus proofs with the implementation, running them in CI alongside conventional tests, and treating proof-model drift as a defect class to be detected is one practical response to that gap, not a complete resolution of it.

The memory-safety claim behind Alcyone is only credible if the unsafe boundary is narrow. In the codebase, unsafe code is confined to platform runtimes, hardware and FFI bindings, and other boundary crates rather than the service and mission logic where most architectural claims live. That confinement is as important as the absence of memory-safety bugs in any one file because it tells the reviewer where the true trust boundary sits in the implementation.

Static and dynamic checks reinforce that boundary. Clippy and other CI gates keep the general code hygiene strict. Interpreter-based undefined-behavior checks are used to inspect risks at the safe/unsafe boundary and in runtime construction paths. Cross-compilation and platform-specific builds verify that service logic does not quietly depend on host-only behavior. These mechanisms are not substitutes for architectural reasoning, but they make the implementation less likely to undermine the architecture's claims.

5.6.2 System-Level Verification in SWIL

System-level evidence comes from four test campaigns. The Alcyone test documentation organizes them as functional verification, adversarial resilience validation, NOS3 integration and SWIL testing, and quantitative resilience assessment. Across those campaigns, the generated test artifact maps 42 test cases and provides direct coverage for 69 of the 89 individual shall-statements (top-level plus sub-requirements).

Adversarial effects enter through the same external boundary that operations use: CCSDS command and telemetry interfaces. Taurus, the adversarial test harness used in Chapter 6, and NOS3 provide the system-level environment for that work, while deterministic metadata capture, seeded randomness, and scenario control preserve repeatability. The architecture is therefore exercised through the paths it claims to defend rather than through hidden debug hooks or artificial shortcuts.

This same SWIL environment supports quantitative resilience measurement. Structured observables, telemetry, event traces, and timing data make it possible to compute quantities such as mean time to detect (MTTD), mean time to recover (MTTR), and mission-function degradation under attack or fault. Taken together, the standards checks, formal analyses, runtime gates, and SWIL experiments provide evidence at the protocol, implementation, and system levels. The empirical assessment in Chapter 6 uses that evidence base.

That evidence is deliberately concentrated rather than uniform. Standards interoperability evidence is strongest for the SDLS and packet-handling paths. Formal analysis is concentrated at authority, routing, serialization, and parser boundaries. System-level testing covers 69 of the 89 shall-statements rather than the full set. The resulting claim is narrower than complete architectural proof: the highest-leverage security and resilience boundaries are supported by explicit and reviewable evidence.

5.7 Comparison with cFS and F' Architectures

cFS and F' are mature, mission-proven frameworks with substantial ecosystem depth and operational heritage. The comparison here asks how the defaults change when a flight software stack is designed for partial compromise rather than broad internal trust.

The central difference is what modularity is expected to do. In cFS and F', modularity primarily serves decomposition, reuse, integration, and reliability engineering. In Alcyone, it also serves containment after compromise, explicit authority control, and measurable enforcement.

cFS and F' are useful foils because they solve real flight-software engineering problems. cFS provides a shared core, mission applications, portable operating-system abstractions, and mature event and telemetry facilities. F' provides stronger component structure through typed ports, topology-driven composition, and model-driven integration tooling. In both cases, however, the security meaning of a component boundary depends heavily on mission-specific deployment choices. A component boundary organizes software, but it does not automatically become an isolation boundary, command-authority boundary, or recovery-authority boundary.

Alcyone moves those boundaries into the default architecture. Services are isolated as runtime tasks or protection domains where the substrate supports it. Internal traffic flows through BUS-mediated topic routing rather than through an implicitly trusted internal fabric. Command ingress is staged through CMD as the sole uplink authority, with authorization remaining mode- and policy-aware. Device authority is also explicit: mission services reach hardware through owner-to-driver routes, and drivers participate as independent bus endpoints instead of being hidden inside mission applications.

Recovery follows the same pattern. In conventional stacks, recovery behavior often emerges from framework services, health-and-safety logic, and mission application code. Alcyone separates the roles: the monitor service observes, FDIR classifies faults, IDS advises on adversarial evidence, and SUP alone decides recovery and mode transitions. On stronger substrates such as seL4, those relationships are reinforced by protection-domain structure and capabilities; on weaker substrates, they remain explicit software boundaries rather than informal

conventions.

The verification posture changes accordingly. cFS and F' rely on mission-specific testing and analysis around mature framework tooling. Alcyone adds requirements traceability, Kani harnesses, Verus proofs for selected invariants, standards-interoperability checks, and SWIL evaluation around the authority and routing boundaries that carry the resilience claim. The comparison is therefore not a feature count. It is a difference in default trust posture: cFS and F' begin from cooperative components integrated into a mission stack, while Alcyone assumes partial compromise and makes internal traffic, authority, recovery, and evidence explicit.

Heritage is the strongest reason to keep choosing cFS. Its deployment record across NASA missions, its mature ecosystem of mission applications and ground tooling, and the cumulative engineering investment in its operational behavior are real assets that an architectural alternative cannot match. A program selecting flight software has rational reasons to prefer the framework with operational evidence over one with architectural arguments alone. The narrower question is whether cFS-class architectures are the only defensible class. Chapter 6 tests that question empirically under common adversarial scenarios.

cFS or F' can be hardened for particular missions; Alcyone instead makes internal trust boundaries, authority separation, and observability first-class architectural objects from the start, changing how compromise propagates under the threat model developed in Chapter 3.

5.8 Summary

Alcyone is a layered, service-oriented flight software system whose basis is documented through linked requirements, threat, architecture, observables, and test artifacts. Command ingress, telemetry, supervision, device access, fault handling, mission applications, and update paths are separated so that authority, containment, and observability remain explicit. The software bus is more than a transport convenience: it is the internal enforcement plane where topic ACLs, transport-verified sender identity, and routing-policy gates make internal traffic something to

validate rather than trust by virtue of being inside the flight image. Recovery and mode authority are concentrated in the supervisor service, with detection roles (MON, FDIR, IDS) advisory and bounded by retry budgets that prevent detector compromise from becoming unbounded recovery action.

Rust strengthens that architecture by making memory safety, ownership, and interface typing part of the enforcement story rather than separate quality goals. The verification and validation basis then builds on top of that safe baseline: Kani harnesses cover bounded properties at parser, routing, and authority boundaries; Verus proofs cover stable structural invariants; interoperability testing validates standards conformance for the most adversary-exposed paths; and SWIL evaluation exercises the full system through the same external interfaces it exposes operationally. Runtime variation enters through bounded platform traits, with seL4 as the deployment-oriented isolation substrate and Linux as the development host. Chapter 6 evaluates whether those design choices produce measurable resilience differences relative to the conventional baseline.

Chapter 6
Experimental Evaluation

Chapter Abstract. Alcyone's architectural choices—compile-time memory safety, process-level isolation, per-command authorization, and bounded supervised recovery—are evaluated for measurable resilience behavior under adversarial stress, with the conventional class of flight software (cFS) as the comparison baseline. Taurus, a SWIL testbed, runs five experiments against a shared Earth-observation mission workload: a comparative architectural benchmark, an Alcyone-focused multi-level functionality measurement, and three sensitivity studies for attack intensity, mission phase, and intrusion-detection configuration. Detection, recovery, functionality retention, and false-positive cost are reported as observable system behaviors rather than as binary pass/fail outcomes.

Chapter Contributions. Four contributions follow. First, it presents Taurus, a SWIL adversarial testbed that drives SPARTA-mapped scenarios through the CCSDS command boundary with no agent on the flight target, supporting repeatable architecture-level experiments under matched mission workload. Second, it adapts Weisman et al.'s functionality-based resilience-measurement protocol to spacecraft systems by defining component, subsystem, and mission-level functionality and reporting cross-level attenuation alongside MTTD, MTTR, blast radius, recovery completeness, and detection coverage. Third, it produces a paired comparative result showing that Alcyone's enforcement surfaces produce measurably different resilience outcomes than the cFS baseline under matched adversarial conditions (Experiment 1), with the multi-level functionality study locating where in the stack mission utility is preserved (Experiment 2). Fourth, it characterizes the Alcyone intrusion detection service (IDS) operating envelope through intensity sweeps, phase-timing study, and detector-mix ablation (Experiments 3–5), treating IDS performance as a tunable engineering trade-off rather than a fixed property of the architecture.

The comparison asks whether the enforcement mechanisms justified earlier produce observ-

able differences when both architectures face the same mission workload, attack content, and measurement conventions. Misses and configuration limits are part of the result, not exceptions to it: a resilience evaluation is useful only if it can report where the architecture produces evidence, where it preserves mission functionality, and where the current detector or scenario set falls short.

6.1 Related Work

6.1.1 Cyber Resilience Measurement and Metrics

Quantitative resilience measurement in cyber-physical systems faces a persistent tension between generality and operational meaning. Segovia et al. [50] survey the CPS resilience literature and identify a convergence toward functionality-based metrics and recovery-centric evaluation, but note that few studies apply these metrics to controlled architectural comparisons. The NIST SP 800-160 Vol. 2 cyber resiliency goals—anticipate, withstand, recover, and adapt (Section 2.3)—provide a useful resilience vocabulary [8], but operationalizing them requires concrete measures: time-to-detect, time-to-recover, and the extent to which mission-relevant functionality is retained during and after attack.

Kott et al. [108] argue that resilience can be treated as a measurable property of system performance under stress, with explicit attention to functionality retention and recovery dynamics. Weisman et al. [109] extend this orientation into a functionality-based resilience-measurement protocol that grounds resilience assessment in observable recovery trajectories and defines a resilience ratio R as the integral of functionality preserved under attack divided by nominal functionality. The evaluation adopts that orientation: metrics are chosen to reflect behaviors that mission operators can observe and that architectures can plausibly influence.

Several assessment frameworks exist in practice—for example, MITRE's Structured Cyber Resiliency Analysis Methodology (SCRAM) scoring approaches [110]—but they are often qualitative or designed for organizational assessment rather than architecture-level measure-

ment. This evaluation emphasizes empirical comparison, with quantitative models [108] used to interpret results rather than to replace measurement.

6.1.2 Adversarial Testing for Space Systems

Space cybersecurity evaluation has historically emphasized policy and process over adversarial demonstration against realistic mission software stacks. NASA IV&V's cyber range provides a concrete counterexample: a virtual environment that spans representative spacecraft and ground components and supports red/blue exercises [46]. Recent security research has demonstrated the feasibility of vulnerability discovery and exploitation against satellite software at scale, including fuzzing and automated analysis in constrained environments [93]. These results motivate an evaluation posture in which compromise is treated as plausible; the central question becomes how architectures detect, contain, and recover rather than whether they can prevent all compromise.

NASA's Operational Simulator for Small Satellites (NOS3) provides the SWIL environment for the campaign [111]. NOS3 integrates hardware simulators, dynamics engines, and time synchronization so that flight software executes against simulated sensors and actuators in a deterministic, repeatable environment, isolating architectural effects from environmental variability.

Adversarial emulation has matured in enterprise security but has seen limited application to spacecraft. Enterprise frameworks typically drive ATT&CK-mapped campaigns from agents installed on target hosts—a model that does not translate to flight software, where the only realistic external attack surface is the command and telemetry boundary. The Aerospace Corporation's SPARTEND project integrates the SPARTA threat taxonomy with autonomous on-board detection against SPARTA-classified threats [112]. The evaluation here takes a complementary direction: a scenario-driven adversarial framework injects SPARTA-mapped external attacks through the CCSDS command boundary, with no agent deployed on the flight target. Additional internal fault interfaces are used only in experiments that model internal

compromise, service failure, or infrastructure stress as controlled resilience stimuli rather than as realistic external-access claims.

6.1.3 Fault Injection and Conventional FSW Testing

The mechanism Taurus uses—injecting controlled stimuli at well-defined interfaces and observing system response—inherits a long lineage from software-implemented fault injection (software-implemented fault injection (SWIFI)). Hsueh et al. [113] survey the canonical taxonomy of fault injection techniques (hardware, software-implemented, and simulation-based), and Carreira et al. [114] demonstrate a representative SWIFI tool that triggers controlled faults via processor exception handlers. Mainline flight-software programs draw on this lineage to validate fault tolerance and FDIR behavior: campaigns inject single-event-upset surrogates, sensor failures, watchdog timeouts, and bus-protocol errors to verify that recovery logic responds as specified, and the disciplined V&V practice described in Chapter 2 (Section 2.1.3) integrates such tests with requirements traceability and configuration control.

Taurus extends this lineage rather than replacing it. The injection mechanism is similar—scripted stimuli applied at architectural boundaries, with effects classified from event logs and telemetry—but the intent and constraint model differ. Conventional fault injection draws from probabilistic fault models (radiation upsets, component degradation) and frequently allows agent-on-target instrumentation because the goal is to characterize random-fault behavior. Adversarial scenario injection draws from threat catalogs (SPARTA techniques mapped to representative kill chains) and forbids agent-on-target instrumentation because the goal is to characterize behavior under stimuli an adversary could realistically deliver through the same interfaces an operator uses. The taxonomy in Section 6.2.2 reflects this distinction: it includes traditional service-failure and sensor-denial primitives drawn from SWIFI practice alongside command-boundary primitives that have no analog in random-fault testing. The metric framework in Section 6.3 is similarly extended—MTTD, MTTR, and recovery completeness map cleanly from fault-tolerance evaluation, while blast radius, detection coverage, and the resilience ratio R are

added to capture concerns specific to adversarial stress.

6.2 Experimental Setup and Methodology

6.2.1 Test Environment

All five experiments share the same mission context: an Earth-observation imaging mission in which the spacecraft receives planning inputs, slews to targets, captures imagery, stores data onboard, and downlinks during contact windows. The observables used across experiments include detection events, recovery markers, mission completion, data returned, telemetry continuity, and functionality over time. The evaluation environment is organized as a layered testbed so that the same architecture can be exercised from unit scope through full-system adversarial campaigns.

The testbed is organized in four layers, each building on the previous:

1. **Unit testing**: individual crate and module tests with harness-provided bus and clock fixtures under deterministic execution. Verifies component logic in isolation.

2. **Integration testing**: multi-service interaction on a Linux host. All services run on a software bus with test harnesses that inject commands and faults programmatically.

3. **System-level SWIL**: NOS3 orchestration with dynamics simulation, hardware simulators, and ground station connectivity. Flight software executes against a full closed-loop environment—sensors receive dynamics-driven data, actuator commands feed back to the dynamics engine, and command and telemetry flow through a realistic CCSDS interface. Alcyone executes within seL4 protection domains under QEMU emulation orchestrated by NOS3; the cFS baseline executes on its native RTOS within the same orchestration framework. The evaluated isolation mechanism is therefore seL4 protection-domain isolation rather than process-level isolation on a commodity host.

4. **Adversarial automation**: scenario-driven fault and attack injection with SPARTA-mapped abilities, automated campaign execution, and metric collection. Adds structured adversarial stress to the SWIL environment.

The experimental benchmarks operate primarily at Layers 3 and 4, where architectural effects on resilience can be measured in a controlled but realistic environment. Layers 1–2 provide the verification foundation on which the SWIL experiments depend.

Layer 3 is built on NASA's Operational Simulator for Small Satellites (NOS3) [111]. NOS3 provides hardware simulators, the 42 dynamics simulator, time synchronization via NOS Engine, and container-based orchestration that allows flight software to execute against simulated sensors and actuators in a deterministic, repeatable closed-loop environment.

Layer 4 is implemented by Taurus, a purpose-built adversarial testbed that extends the Layer 3 environment with scenario automation, repeatable fault injection, and experiment telemetry capture. Its role is not to replace the SWIL stack but to add the functions needed for controlled comparative experiments: batch scenario definition, scheduled command and fault dispatch, parameterized attack profiles, controlled internal stressors, deterministic artifact collection, and post-run metric aggregation. In practice, Taurus treats the simulator, flight software, ground interface, device models, communications path, and observability services as one orchestrated experiment system behind a single host-side control point. That organization allows the same scenario definition to be replayed across architectures and operating conditions while preserving timing control, mechanism traceability, and reproducible outputs for later analysis.

6.2.2 Attack Injection Framework

The injection framework distinguishes between external adversary stimulus and controlled internal stressors. External attack scenarios enter the system under test through the same interfaces legitimate commands use. Internal fault and infrastructure mechanisms are used only when the experiment explicitly models internal compromise, service failure, or degraded operating conditions. Three constraints govern the framework:

1. No external adversary action bypasses the CCSDS command ingress boundary. External attacks arrive through the same interface as operator commands.

2. No adversary agent or process runs on the flight software target. The target executes only its own code; external attack stimulus remains external.

3. Effects are classified only when they appear in event logs or telemetry. If an attack has no observable consequence, the run records either ineffective stimulus or insufficient instrumentation.

Each injection is time-tagged, attributed to a specific SPARTA technique and requirement where applicable, and logged in the event stream with sufficient detail to reconstruct the injection timeline post hoc. External adversary campaigns are constructed from command-boundary and protocol-layer mechanisms; internal fault and infrastructure mechanisms are reserved for experiments that explicitly study containment, degraded operation, or recovery under controlled stress. Higher-level kill chains compose these primitives into multi-stage campaigns drawn from public incidents and internal threat analysis. The complete scenario suite used across the experiments is defined in Section 6.2.4 (Table 6.1).

6.2.3 Experiment Set

Five experiments test whether the architecture improves resilience, how that improvement can be measured, and where it strengthens, weakens, or fails under operational and configuration changes. **Experiment 1 (Comparative Architectural Resilience)** is the primary systems experiment, comparing Alcyone and cFS under matched mission workload and adversarial conditions. **Experiment 2 (Quantitative Measurement of Cyber-Resilience)** is an Alcyone-focused measurement experiment that adapts functionality-based resilience measurement to component, subsystem, and mission levels. **Experiment 3 (Attack Intensity Threshold Sweeps)** varies attack magnitude to characterize the operating envelope: detection reliability, time to detection, and detection confidence. **Experiment 4 (Attack Timing by Mission Phase)**

holds attack content constant while varying the injection point across imaging, coast, and downlink phases. **Experiment 5 (IDS Configuration Sensitivity and Ablation)** varies detector windows, thresholds, and detector subsets to measure latency, false positives, and per-detector contribution.

Each experiment uses the same Earth-observation mission context and testbed stack, so differences in outcomes are attributable to the architecture, attack condition, mission phase, or detector configuration being varied rather than to changes in mission workload.

6.2.4 Shared Scenario and Run Structure

The thirteen-scenario comparative core is reused across the experiments. Ten f-series scenarios target single architectural mechanisms; three a-series scenarios compose multi-stage adversary campaigns drawn from public incident analysis. Table 6.1 defines each scenario, the SPARTA technique it exercises, and the requirements it tests.

Each scenario produces event evidence, telemetry time series, and recovery or degradation traces. Where an experiment defines the needed baseline and repeated trials, those traces support MTTD, MTTR, and resilience-ratio calculations; otherwise they support scorecard evidence and bounded case interpretation.

For the evaluation, a *scenario* is one attack or fault definition, a *campaign* is a batch execution plan over one or more scenarios, and a *run* is one execution of a scenario under one architecture and profile. The orchestrator loads campaign definitions, applies architecture-specific profiles, dispatches scheduled commands or faults, and collects the resulting artifacts. Where the stimulus is transport-level—for example, sensor drops, command floods, or service failures—the same scenario can be replayed across architectures. Where a scenario uses architecture-specific command surfaces or simulator control paths, paired variants preserve the same adversarial intent on different implementation surfaces.

Each campaign run produces a standard artifact set: raw CCSDS packet captures, decoded event logs, housekeeping telemetry, scenario metadata, run metadata, and a derived report

Table 6.1 Comparative scenario suite reused across the experiments. The f-series scenarios are single-mechanism attacks; the a-series scenarios are multi-stage adversary campaigns. Dashes (—) in the SPARTA column mark scenarios that exercise architectural enforcement surfaces SPARTA does not enumerate at this granularity (bus spoofing, telemetry manipulation, forced mode transition); these remain mapped to the requirement identifiers they test.

ID	Mechanism	SPARTA	Requirement
f1	Sustained high-rate command flood	REC-0002	CR-5, HS-6
f2	Valid-syntax command with invalid authorization	EX-0001	CR-1, CR-8
f3	Replay or reordering of captured commands	EX-0001	CR-1, CR-3
f4	Withheld or delayed sensor data	EX-0014	TG-1, HS-7
f5	Targeted service crash via fault-injection command	—	HS-1, HS-11
f6	Internal bus message with spoofed source	—	TG-1, TG-6
f7	Telemetry path manipulation	—	TG-1, TG-6
f8	Forced mode transition or recovery interference	—	HS-1, CR-12
f9	Compound blind/destabilize (sensor drop and command flood)	composite	aggregate
f10	Compound drain/corrupt (queue saturation and replay)	composite	aggregate
a1	Reconnaissance leading to exploit	multi-stage	aggregate
a2	Coordinated deny-recovery campaign	multi-stage	aggregate
a3	Ground supply-chain compromise affecting commands	IA-0004	aggregate

or scorecard. Each experiment specifies its scenario subset, repetition count, and analysis method.

6.2.5 Common Analysis and Validity Rules

The common rules apply only to outcomes shared across all five experiments, since the experiments use different trial structures. Every run is first reduced to observable outcomes: execution status, injection interval, detection verdict, MTTD when a qualifying detection exists, recovery marker and MTTR when applicable, functionality signal when defined, and artifact completeness. Aggregate statistics are reported only for experiment cells with repeated runs; otherwise the result is reported as scorecard evidence or bounded case evidence.

The multi-level resilience analyses—functionality traces, area-under-curve ratios, matched nominal/attacked cohorts, and cross-level attenuation gaps—belong to Experiment 2 rather than to the common methodology for the chapter. Additional statistical tests are reserved for larger repeated-campaign comparisons rather than imposed on every experiment.

Validity rules constrain what constitutes an acceptable experimental run:

- Repeatability is mandatory: identical scenario and firmware must yield the same detection outcome and recovery classification across runs.
- One primary stressor is varied at a time unless the experiment is explicitly designed as a compound-stressor test.
- Manual intervention during a run invalidates it.
- Runs with missing logs, metadata, or silent failures (no events despite known stimulus) are excluded.

Trial counts vary by experiment. Experiments 3, 4, and 5 use three trials per cell so detection probability and median MTTD can be reported with within-cell variance. Experiment 1 uses one trial per paired scenario because the SWIL substrate executes deterministically under fixed

seeds and scheduling, with the repeatability rule above verified during campaign development against selected cells. Experiment 1 results are therefore reported as paired scorecard evidence rather than as cohort statistics: the comparative-architecture claim rests on the per-scenario direction and magnitude of the Alcyone-versus-cFS difference, not on within-cell distributional inference. Section 6.9.4 addresses the resulting sample-size question directly.

6.3 Resilience Metrics

The metrics chosen here are operationally interpretable, measurable in SWIL, and sensitive to architectural differences in isolation, integrity enforcement, and recovery behavior. Each metric is mapped in Table 6.2 to the NIST SP 800-160 Vol. 2 cyber resilience goal it primarily addresses.

Mean time to detect (MTTD) is the elapsed time between attack initiation and detection by the system under test, where detection is an observable event—an alert, a mode transition, or a logged integrity failure—emitted by an instrumentation point. Because detection definitions are architecture-dependent, the evaluation specifies a detection event schema per system so that MTTD is comparable across implementations. For Alcyone, detection events include health-state transitions reported by the supervisor service, integrity-check failures at the command service, and anomaly reports from the monitoring service. For cFS, equivalent events are identified from the Event Services subsystem and mapped to the same schema.

Recovery is measured against an intentionally resilience-oriented definition: not perfect restoration, but restoration of mission-critical control. Mean time to recover (MTTR) is the elapsed time from detection until the system reaches an acceptable operating state—either nominal operation or a pre-defined degraded mode that preserves core control and safety functions. Recovery markers include service restart events, mode stabilization (sustained nominal or degraded mode for a configurable hold period), and resumption of telemetry output at expected rates. Where recovery requires operator intervention in one architecture but not

another, that difference is reported explicitly.

Functionality retention under attack is measured as the fraction of defined mission-relevant capabilities maintained during and after the attack interval. Measurement requires defining a functionality model for the system under test [108]; the model is scenario-specific and is reported alongside results to bound claims of generality. The evaluation adopts the functionality-based approach of Weisman et al. [109]: a time-varying functionality metric $F(t)$ represents mission-relevant performance, and a resilience ratio is computed as the integral of functionality under attack divided by nominal functionality,

$$R = \frac{\int_0^T F_{\text{attacked}}(t)\, dt}{\int_0^T F_{\text{nominal}}(t)\, dt} \qquad \text{(Equation 6.1)}$$

where $R \in [0, 1]$: $R = 1.0$ indicates the attack had no effect on functionality, and $R = 0.0$ indicates complete loss for the scenario duration. $F(t)$ is defined in terms of observable mission outputs rather than internal software metrics. To preserve cross-architecture comparability, the component metrics are defined at the mission-function level common to both baselines rather than at the level of implementation-specific services. The component metrics contributing to $F(t)$ are **command throughput** (fraction of valid commands successfully processed per unit time), **telemetry continuity** (fraction of expected telemetry packets produced), **sensor-to-actuator latency** (end-to-end pipeline delay from sensor ingress to actuator command), and **service availability** (fraction of monitored services in a healthy state). A composite functionality score combines these with scenario-specific weights:

$$F(t) = w_{\text{cmd}} \cdot F_{\text{cmd}}(t) + w_{\text{tlm}} \cdot F_{\text{tlm}}(t) + w_{\text{lat}} \cdot F_{\text{lat}}(t) + w_{\text{avail}} \cdot F_{\text{avail}}(t) \qquad \text{(Equation 6.2)}$$

Weights are declared per experiment scenario and held constant across compared runs. Functionality-over-time curves combined with area-under-curve comparison provide a continuous measure sensitive to both the severity and the duration of impact, avoiding the limitations of

binary pass/fail resilience assessment.

Blast radius measures the spatial extent of adversarial impact: the count of distinct mission functions or normalized subsystem roles whose nominal operation was disrupted before containment. For each scenario s,

$$\text{BR}(s) = |\{f \in F : \text{impact}(f, s) = \text{true}\}| \quad \text{(Equation 6.3)}$$

where F is the set of monitored mission functions or normalized subsystem roles used consistently across both architectures for that scenario. The metric directly measures the effectiveness of isolation boundaries: an architecture with strong inter-service isolation should exhibit lower blast radius than one where compromise propagates freely across a shared address space.

Recovery completeness is the fraction of pre-attack system capability restored after recovery completes:

$$\text{RC}(s) = \frac{|\{f \in F : \text{restored}(f, s) = \text{true}\}|}{|\{f \in F : \text{impacted}(f, s) = \text{true}\}|} \quad \text{(Equation 6.4)}$$

Range is 0 (no recovery) to 1 (full recovery). Together with MTTR, this metric distinguishes architectures that recover quickly but incompletely from those that achieve full restoration.

Detection coverage is the fraction of executed adversarial scenarios that produced at least one detection event:

$$\text{DC} = \frac{|\{s \in S : \text{detected}(s) = \text{true}\}|}{|S|} \quad \text{(Equation 6.5)}$$

False positive rate (FPR) is the rate of spurious detection events during non-adversarial operation, measured during baseline operational runs with no adversarial injection:

$$\text{FPR} = \frac{\text{spurious alerts}}{\text{nominal operating hours}} \quad \text{(Equation 6.6)}$$

High FPR degrades operational confidence and can exhaust recovery budgets. Together, DC

and FPR characterize the Anticipate goal of the NIST SP 800-160 Vol. 2 framework: a system that detects threats reliably without generating false alarms that trigger unnecessary safe-mode transitions.

Table 6.2 maps each metric to the NIST SP 800-160 Vol. 2 resilience goals it primarily addresses.

Table 6.2 Mapping of resilience metrics to NIST SP 800-160 Vol. 2 cyber resilience goals.

Metric	**Anticipate**	**Withstand**	**Recover**	**Adapt**
MTTD	✓	✓		
MTTR			✓	
Blast Radius		✓		
Recovery Completeness			✓	
Detection Cov.	✓			✓
False Positive	✓			
Resilience ratio R	✓	✓	✓	

6.4 Experiment 1: Comparative Architectural Resilience

6.4.1 Design

Experiment 1 is the primary comparative systems experiment. The independent variable is the flight software architecture—Alcyone or cFS. The controlled variables are the mission workload, attack timing and content, simulation environment, and ground interaction model. The dependent variables are operationally meaningful resilience outcomes: whether the attack was detected, how long detection took, whether recovery occurred autonomously, how long recovery took, how much mission completion was preserved, how much data was returned, and how the system moved through operating modes.

The experiment targets the architectural factors discussed in Chapters 3–5: memory safety, isolation boundaries, and integrity and authorization enforcement at internal interfaces. It does not isolate each factor independently. Instead, it compares two architectural packages whose

defaults embody different design assumptions about trust, containment, and recovery. The goal is not to declare a universal winner; it is to test whether the security-oriented design choices integrated into Alcyone produce measurably different resilience behavior than the conventional class represented by cFS. A favorable result for Alcyone is not simply that it finishes the mission. It is earlier evidence, smaller blast radius, explicit rejection, and equal or better mission functionality under the same attack intent.

Both architectures run their default security posture. For cFS, this means the standard cFE core—Executive Services, Software Bus, Event Services, Table Services, Time Services—with representative mission applications and no additional cyber-hardening components, since cFS has no standardized hardening layer (Section 5.7). Alcyone runs its service-oriented architecture with command-validation, bus-policy, observability, and bounded-recovery mechanisms enabled. Table 6.3 summarizes the key architectural differences between the two baselines.

Table 6.3 Architectural comparison of the two primary flight software baselines used in Experiment 1.

Characteristic	cFS	Alcyone
Language	C	Rust (`no_std`)
Memory safety	Manual	Compile-time enforcement
Isolation model	Shared address space	Service isolation (seL4 protection domains, evaluated under QEMU emulation)
IPC mechanism	Software Bus (SB)	Typed message bus with ACLs
Authorization	None (cFE default)	Per-command, mode-aware
Recovery	App restart (ES)	Bounded restart with escalation

Experiment 1 runs the thirteen-scenario comparative core defined in Section 6.2.4 (Table 6.1). The f-series scenarios (f1–f10) provide the fixed matrix of single-mechanism attacks, with the f9 and f10 compound cases preserving attack intent across the two implementation surfaces. The a-series scenarios (a1–a3) exercise recovery under chained, multi-stage effects. Paired scenario cells provide the comparative basis by sharing mission workload, injection timing, and observability requirements; a compact staged form of the same scenarios is run as an escalating

sequence to validate that the comparative logic holds under reduced campaign runtime.

Attack response behavior is evaluated through detection latency, containment markers (whether compromise effects propagate across components), and whether the system enters an acceptable degraded state. The benchmark looks specifically for differences in whether malformed input causes a crash versus a logged rejection, whether compromise of one component allows interference with others, and whether unauthorized commands are silently accepted, silently dropped, or explicitly rejected with audit evidence.

Recovery characteristics are evaluated through MTTR and a qualitative recovery-mode classification: restart, reconfiguration, safe mode, or irrecoverable loss of function for the scenario. Operator intervention remains part of the result when one architecture requires it and the other does not; the comparison does not normalize that difference away.

Degraded-mode operation is evaluated by functionality retention during attack and during recovery. The evaluation treats graceful degradation as a positive outcome when it preserves command authority and safety constraints, even when it sheds non-essential mission functions.

The interpretation ties measured differences back to architectural features: **memory safety** (whether failures appear as crashes or exploits versus rejected inputs or contained faults), **isolation** (whether compromise effects are bounded to a single component or propagate across the address space), **zero-trust communication** (whether internal message manipulation is detected and rejected by bus-level access control, or accepted silently), and **bounded recovery** (whether the architecture supports automatic, bounded recovery versus requiring ground intervention).

6.4.2 Results

The Experiment 1 campaign produced 13 paired scenarios across the comparative core suite. The campaign analyzer reduces each run's event streams, metadata, and telemetry captures into per-scenario scorecards with detection, MTTD, MTTR, resilience ratio, blast radius, recovery completeness, and accepted-adversary-action fields. Table 6.4 reports detection, latency,

functionality, and containment for each architecture under matched mission workload, injection timing, and adversarial intent. Table 6.5 reports the corresponding recovery metrics.

Table 6.4 Per-scenario scorecard for Experiment 1, paired Alcyone vs. cFS. Det. indicates qualifying onboard detection; MTTD is in seconds; R is the resilience ratio; BR is blast radius (mission functions impacted before containment).

	Alcyone				**cFS**			
Scenario	Det.	MTTD	R	BR	Det.	MTTD	R	BR
a1-recon-exploit	✓	0.300	0.970	1	—	—	0.780	3
a2-deny-recovery	✓	1.000	0.910	2	—	—	0.620	3
a3-ground-supply-chain	✓	2.500	0.930	2	—	—	0.580	4
f1-cmd-flood	✓	1.800	0.940	1	—	—	0.720	4
f2-unauthorized-cmd	✓	0.163	1.000	0	—	—	0.960	1
f3-cmd-replay	✓	0.220	1.000	0	✓	6.800	0.910	2
f4-sensor-drop	✓	0.087	0.980	1	—	—	0.890	2
f5-app-kill	✓	1.200	0.910	2	✓	7.200	0.760	3
f6-bus-spoof	✓	0.470	0.970	1	—	—	0.740	4
f7-tm-manipulation	✓	0.910	0.990	1	✓	9.400	0.860	2
f8-force-reduced-ops	—	—	0.890	2	—	—	0.400	4
f9-compound-blind-destabilize	✓	0.700	0.860	2	—	—	0.550	4
f10-compound-drain-corrupt	✓	0.620	0.880	2	—	—	0.610	3

The architectural separation is large in detection coverage and blast radius and consistent in resilience ratio. Alcyone produced a qualifying onboard detection in twelve of thirteen tested scenarios (12/13), giving detection coverage $DC = 0.923$. The exception is the force-reduced-operations scenario, where the supervisor blocks unauthorized demotion and preserves bounded function but the monitoring path does not emit a qualifying evidence event. The cFS baseline produced an equivalent detection event in three scenarios: command replay (`f3-cmd-replay`, MTTD 6.800 s), application kill (`f5-app-kill`, MTTD 7.200 s), and telemetry manipulation (`f7-tm-manipulation`, MTTD 9.400 s), giving $DC = 0.231$. The remaining ten scenarios completed without a qualifying cFS detection event despite, in several cases, observable mission impact reflected in the resilience-ratio and blast-radius columns. Across detected

Table 6.5 Recovery scorecard for Experiment 1. MTTR is median time to reach the scenario's acceptable operating state, in seconds; RC is recovery completeness, the fraction of impacted mission functions restored after recovery.

Scenario	Metric	Alcyone	cFS
a1-recon-exploit	MTTR (s)	6.0	50.0
	RC	0.960	0.620
a2-deny-recovery	MTTR (s)	22.0	95.0
	RC	0.880	0.450
a3-ground-supply-chain	MTTR (s)	18.0	80.0
	RC	0.900	0.420
f1-cmd-flood	MTTR (s)	12.4	44.0
	RC	0.950	0.550
f2-unauthorized-cmd	MTTR (s)	0.4	0.0
	RC	1.000	0.900
f3-cmd-replay	MTTR (s)	1.1	18.0
	RC	1.000	0.720
f4-sensor-drop	MTTR (s)	30.0	35.0
	RC	0.980	0.700
f5-app-kill	MTTR (s)	24.0	70.0
	RC	0.880	0.520
f6-bus-spoof	MTTR (s)	3.5	60.0
	RC	0.960	0.450
f7-tm-manipulation	MTTR (s)	6.0	28.0
	RC	0.980	0.650
f8-force-reduced-ops	MTTR (s)	18.0	180.0
	RC	0.880	0.300
f9-compound-blind-destabilize	MTTR (s)	15.0	85.0
	RC	0.840	0.400
f10-compound-drain-corrupt	MTTR (s)	12.0	72.0
	RC	0.860	0.460

Alcyone scenarios, mean MTTD was 0.83 s; cFS, considering only the three detected scenarios, averaged 7.80 s. Mean resilience ratio was 0.941 for Alcyone and 0.722 for cFS. Mean blast radius was 1.3 mission functions for Alcyone and 3.0 for cFS. The recovery scorecard shows the same pattern after detection: Alcyone's mean MTTR was 13.0 s with mean recovery completeness $RC = 0.928$, while cFS averaged 62.8 s and $RC = 0.549$.

Authority-boundary scenarios produce the cleanest separation. The unauthorized-command and command-replay scenarios (`f2-unauthorized-cmd`, `f3-cmd-replay`) yielded $R = 1.000$ and blast radius 0 for Alcyone, with MTTD under 0.25 s and a recovery interval bounded below 0.5 s. The CMD service rejected the commands before BUS routing and emitted explicit authority-denial events. The cFS baseline, lacking per-command authorization in the default cFE configuration, accepted three unauthorized actions in the `f2` scenario; the operator-visible mission impact had to be reconstructed post hoc from telemetry rather than surfaced at the command boundary.

Bus-spoofing and command-flood scenarios show containment by isolation rather than by detection alone. For `f1-cmd-flood` and `f6-bus-spoof`, both architectures completed the workload, but Alcyone bounded blast radius to a single mission function (queue pressure shed by source-aware BUS ACLs) while cFS exhibited a blast radius of four—telemetry continuity, command throughput, and two service-availability degradations all propagated through the shared software bus. The MTTR difference is similarly large: 12.4 s for the Alcyone `f1` cell versus 44.0 s for the cFS cell, reflecting that recovery in Alcyone is centralized in the supervisor's bounded action set while cFS relies on per-application restart through Executive Services.

Compound and supply-chain scenarios stress the supervisor's bounded-recovery authority. The F9 compound blind-destabilize, F10 compound drain-corrupt, and A3 ground supply-chain scenarios produce multi-stage adversary effects requiring coordinated recovery. Alcyone preserved $R \geq 0.86$ across these; cFS preserved R between 0.55 and 0.61. The difference reflects the SUP service's bounded restart and mode-transition authority: recovery actions are issued from a single auditable point rather than distributed across applications that can

themselves be compromised.

Under matched mission workload, attack content, and observability conventions, Alcyone produces measurably different resilience outcomes than the cFS baseline. The difference is not uniformly distributed across attack classes. Authority-boundary attacks produce the largest separation; resource-pressure attacks separate primarily in containment rather than in detection latency; compound multi-stage attacks separate in recovery completeness and blast radius. Each pattern is traceable to a specific architectural enforcement surface introduced in Chapter 5.

6.5 Experiment 2: Quantitative Measurement of Cyber-Resilience

6.5.1 Design

Where does Alcyone absorb degradation before it becomes mission consequence? Experiment 2 measures internal propagation rather than architectural comparison. It adapts the Weisman et al. functionality-based resilience-measurement protocol [109] to spacecraft systems by defining a multi-level functionality framework and computing resilience ratios at each level.

Functionality is defined at three levels. At the component level, $F_{\text{comp}}(t)$ captures local service and device health: device health states, service liveness, sensor validity, and the bounds on communication and bus behavior. At the subsystem level, $F_{\text{sub}}(t)$ is expressed in operational terms: attitude performance, command acceptance, telemetry continuity, storage and downlink throughput, and image-collection progress. At the mission level, $F_{\text{miss}}(t)$ captures scenario-specific mission accomplishment rather than a single criterion reused across all attacks. Sensor-denial cases emphasize pointing tolerance and target collection; command-boundary cases emphasize preservation of the commanded plan and rejection of unauthorized state change; command-flood cases emphasize command-window utility, contact utilization, and data return; compound cases combine target collection, returned data, and sustained mission-capable mode.

For each level k, the resilience ratio is computed as an area-under-curve ratio:

$$R_k = \frac{\int_0^T F_k^{\text{attacked}}(t)\,dt}{\int_0^T F_k^{\text{nominal}}(t)\,dt} \qquad \text{(Equation 6.7)}$$

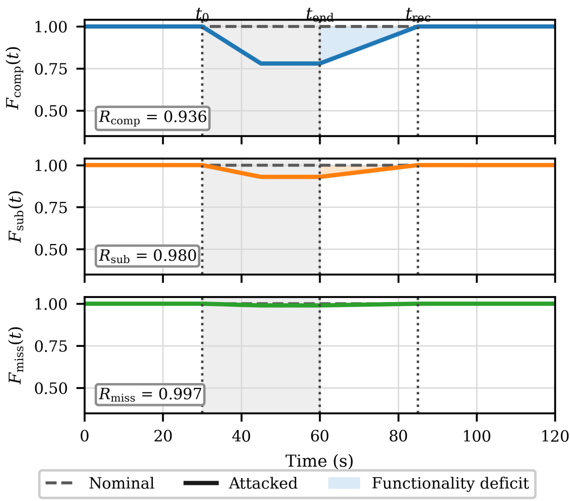
In this model, stronger degradation at lower levels than at the mission level indicates that the architecture absorbs, isolates, or recovers before cyber effects propagate upward. The attenuation pattern $R_{\text{comp}} \leq R_{\text{sub}} \leq R_{\text{miss}}$ is therefore treated as expected evidence of cross-level damping rather than as an invariant. The pattern identifies *where* in the stack resilience was generated and *how strongly* that resilience damped mission impact.

Experiment 2 uses four Alcyone cells from the Experiment 1 attack suite: sensor drop as a localized perception disturbance, unauthorized command as an authority-boundary attack, command flood as a resource-pressure attack, and compound blind/destabilize as a multi-stage adversary effect. For each cell, the analysis computes per-level resilience ratios for matched nominal and attacked traces over a 120 s horizon, then reports an attenuation gap, $\Delta = R_{\text{miss}} - R_{\text{comp}}$.

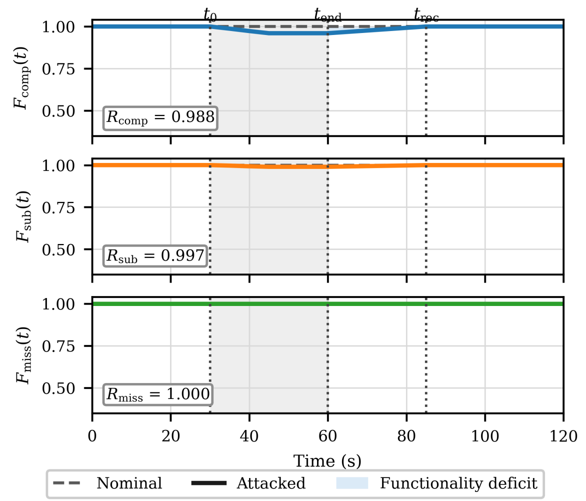
6.5.2 Results

Experiment 2 explains consequence across levels. The campaign pairs nominal and attacked Alcyone runs by runtime, profile, injection phase, attack profile, and trial index. For each matched cell, the analysis extracts component-, subsystem-, and mission-level functionality traces and computes the area-under-curve ratio in Equation 6.7. The reporting unit is a matched cohort, not an individual run.

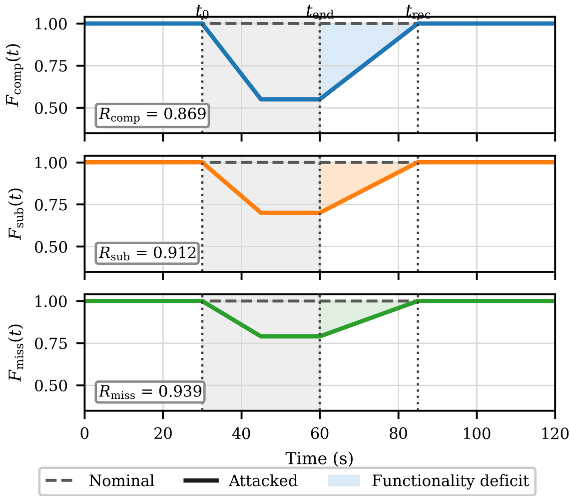
Figure 6.1 shows the time-series and integral interpretation for each of the four scenarios. In each panel the dashed line is the nominal functionality trajectory, the solid line is the attacked trajectory, and the shaded region is the functionality deficit integrated into the numerator-denominator ratio in Equation 6.7. A short, deep disturbance and a long, shallow disturbance can therefore produce different resilience ratios even when their worst instantaneous functionality is similar.



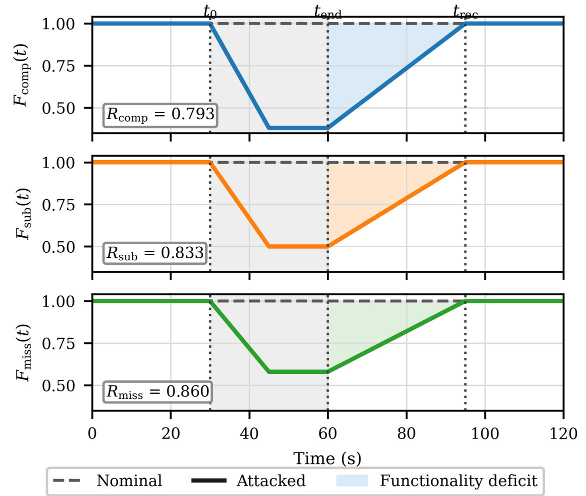
(a) Sensor drop ($R_{\text{miss}} = 0.997$, $\Delta = +0.061$): local perception disturbance damped before mission consequence.



(b) Unauthorized command ($R_{\text{miss}} = 1.000$, $\Delta = +0.012$): rejected at the CMD-service authority boundary.



(c) Command flood ($R_{\text{miss}} = 0.939$, $\Delta = +0.070$): bus-policy isolation contains queue pressure to a single mission function.



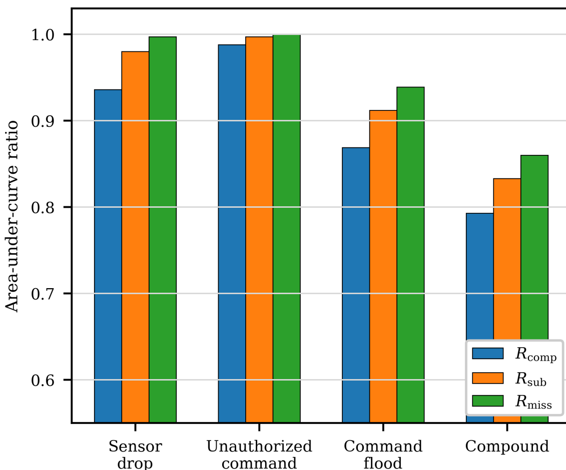
(d) Compound blind/destabilize ($R_{\text{miss}} = 0.860$, $\Delta = +0.067$): supervisor-mediated recovery limits mission impact despite multi-stage local disturbance.

Figure 6.1 Experiment 2 functionality-over-time traces for the four Alcyone scenarios. Each panel shows component-, subsystem-, and mission-level functionality traces, with shaded regions indicating the integrated functionality deficit relative to the nominal trajectory.

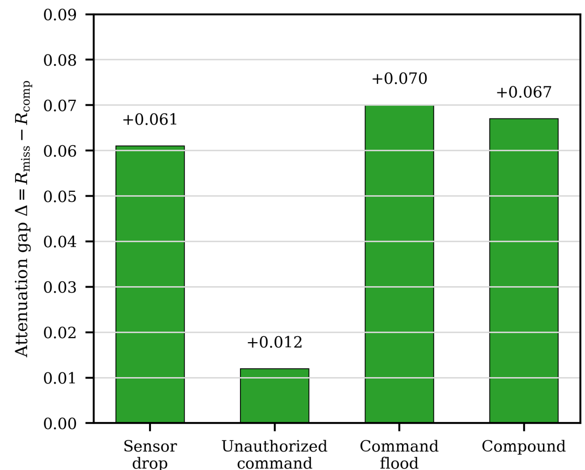
Table 6.6 reports the area-under-curve resilience ratios at component, subsystem, and mission levels for all four Alcyone scenario cells. Figure 6.2 visualizes the same result as level-specific ratios and attenuation gaps.

Table 6.6 Experiment 2: Alcyone area-under-curve resilience ratios at component, subsystem, and mission levels. Δ is the attenuation gap $R_{\text{miss}} - R_{\text{comp}}$ over the 120 s evaluation horizon.

Scenario	R_{comp}	R_{sub}	R_{miss}	Δ
Sensor drop	0.936	0.980	0.997	+0.061
Unauthorized command	0.988	0.997	1.000	+0.012
Command flood	0.869	0.912	0.939	+0.070
Compound blind/destabilize	0.793	0.833	0.860	+0.067



(a) Area-under-curve ratios by level.



(b) Attenuation gaps, $\Delta = R_{\text{miss}} - R_{\text{comp}}$.

Figure 6.2 Experiment 2 summary plots. Positive attenuation means that mission-level functionality preserved more area than component-level functionality.

Alcyone exhibits positive attenuation in every tested scenario. The sensor-drop cell is the clearest localized case: component-level functionality falls to $R_{\text{comp}} = 0.936$, while mission-level functionality remains at $R_{\text{miss}} = 0.997$, a gap of $+0.061$. Command flood and compound blind/destabilize cases produce larger local disturbance, but the same ordering remains: $R_{\text{comp}} \leq R_{\text{sub}} \leq R_{\text{miss}}$. That pattern is the operational signature of a system that absorbs and isolates cyber effects before they propagate to mission consequence.

The attenuation gap identifies whether component-level impairment is contained by the service, bus, supervisor, and mission-control boundaries before mission-level functionality loses area under the curve.

The weighting set is recorded in the run metadata and reproduced alongside the time-series functionality traces. Because R is a composite measure, different mission concepts may reasonably weight data return, pointing tolerance, command acceptance, or telemetry continuity differently. Experiment 2 therefore reports the level-specific curves and attenuation gaps, not only the final scalar.

6.6 Experiment 3: Attack Intensity Threshold Sweeps

6.6.1 Design

Experiment 3 characterizes the Alcyone detection envelope as a function of attack magnitude rather than repeating the architectural comparison. The hypothesis is that detection performance is not intensity-invariant: each attack class should have a threshold region below which detection is weak or inconsistent and above which detection becomes reliable and faster.

The sweep covers command-flood rate, sensor-drop duration, unauthorized-command burst count, and bus-spoof topic deviation. The independent variable is attack magnitude; the dependent variables are detection probability, MTTD, and detection confidence. Each sweep point is repeated three times under the same mission workload and detector configuration, and the selected threshold values define the attack magnitudes used in Experiment 5's configuration studies.

6.6.2 Results

The Experiment 3 campaign swept four attack-magnitude parameters across Alcyone: command-flood rate (10–2,000 packets/s), sensor-drop duration (1–60 s), unauthorized-command burst count (1–20), and bus-spoof topic deviation (1%–100%). Each sweep point ran three trials.

Table 6.7 reports the selected reliable threshold for each detector family, the detection probability and median MTTD at that threshold, and the detection-probability range across the full sweep.

Table 6.7 Experiment 3: detection envelope across four attack-magnitude sweeps. Reliable threshold is the selected sweep point used for downstream configuration studies. Detection range reports P_{detect} at the lowest and highest sweep points.

Attack family	**Sweep parameter**	**Threshold**	P_{detect}	**MTTD (s)**	**Range**
Command flood	packets/second	1,000	0.990	1.800	0.020 → 1.000
Internal bus spoofing	topic deviation (%)	50	0.990	0.500	0.050 → 1.000
Sensor drop	drop duration (s)	15	0.980	8.300	0.050 → 1.000
Unauthorized command	burst count	5	1.000	0.160	0.550 → 1.000

The sweep results show three distinct evidence regimes rather than one generic IDS sensitivity curve. Unauthorized-command bursts reach reliable detection at burst count 5 with median MTTD of 0.160 s because the stimulus crosses an explicit CMD authority boundary. Detection does not depend on a statistical signal accumulating against nominal traffic.

Resource and protocol attacks require a larger observable signal. Command flood becomes reliable at 1,000 packets/s, and bus spoofing becomes reliable at 50% topic deviation. Below those transition regions, the evidence stream is too close to nominal variation for the selected detector configuration to produce qualifying evidence.

Stale-data detection is duration-bound. The 15 s reliable threshold and 8.300 s median MTTD reflect the confirmation window needed to distinguish adversarial withholding from transient sensor timing. The latency is therefore part of the detector design, not merely slow event handling.

The practical result is an operating envelope. Low-intensity misses are not anomalies in the experiment; they define where this IDS configuration stops making a defensible detection claim. Experiment 5 uses those thresholds to evaluate whether the selected operating point balances detection probability, latency, and false-positive cost.

6.7 Experiment 4: Attack Timing by Mission Phase

6.7.1 Design

Experiment 4 isolates mission phase as the explanatory variable by injecting the same attack content during imaging, coast, and downlink. The hypothesis is that mission impact varies across phases because each phase has different resource demands, data rates, and nominal behavior, even when detector latency remains stable.

The design uses three representative attack families: unauthorized command, command flood, and sensor drop. Each family is injected during all three phases, producing nine phase cells with three trials per cell. Holding attack magnitude fixed separates phase-dependent mission consequence from detector sensitivity.

6.7.2 Results

The Experiment 4 phase-timing campaign produced 27 runs across the nine attack-phase cells. Table 6.8 reports detection counts, latency range, mean resilience ratio, and operational effect for each cell.

Experiment 4 separates local security outcome from mission consequence. All 27 runs produced a qualifying detection, and no adversary action was accepted, so mission phase did not materially change whether the attack was recognized or blocked. Latency remained attack-family-specific: authority violations were rejected at ingress, sensor drops were detected through stale-data evidence, and command floods required a sustained rate signal.

Mission consequence was phase-dependent. Unauthorized-command bursts were nearly invariant because rejection occurred before the command interacted with phase-specific resources. Command flood was most costly during downlink, when the command path and operator timeline were active. Sensor drop was most costly during imaging, where pointing tolerance directly affected mission utility.

The operational lesson is that detector correctness is not the same as resilience cost.

Table 6.8 Experiment 4: phase-timing results for three attack families injected during three mission phases. Detection counts are over three trials per attack-phase cell; no cell accepted adversary actions.

Attack	Phase	Det.	MTTD (s)	Mean R	Operational effect
Unauthorized command	Imaging	3/3	0.160–0.179	0.992	Command rejected; image plan preserved.
Unauthorized command	Coast	3/3	0.160–0.179	1.000	Command rejected; no mission-level consequence.
Unauthorized command	Downlink	3/3	0.160–0.179	0.996	Command rejected; minor operator-timeline effect.
Command flood	Imaging	3/3	1.800–2.016	0.940	Command sequence disrupted; supervisor shifts to safe imaging plan.
Command flood	Coast	3/3	1.800–2.016	0.980	Flood detected and shed; no critical timeline impact.
Command flood	Downlink	3/3	1.800–2.016	0.850	Command path saturated during operator window; downlink schedule disrupted.
Sensor drop	Imaging	3/3	0.087–0.097	0.950	Redundant reference engaged; pointing tolerance briefly exceeded.
Sensor drop	Coast	3/3	0.087–0.097	0.990	Stale-data evidence generated; no mission-relevant pointing requirement.
Sensor drop	Downlink	3/3	0.087–0.097	0.980	Antenna pointing degraded but remains within link margin.

Recovery budgets, operator-review thresholds, and degraded-mode transitions should depend on when a cyber event occurs, not only on what class of event was detected.

6.8 Experiment 5: IDS Configuration Sensitivity and Ablation

6.8.1 Design

Experiment 5 treats IDS settings as tunable engineering choices rather than hidden defaults. It separates mechanism selection from parameter tuning by measuring how windows, thresholds, and detector subsets affect latency, sensitivity, false positives, and per-detector contribution.

Experiment 5 isolates intrusion-detection configuration as the explanatory variable. Window duration, detector thresholds, and detector mix vary; mission workload, attack timing, and phase remain constant. The study asks three narrower questions: how much temporal evidence the IDS accumulates before reporting a detection, how sensitive each detector can be before it consumes supervisor attention and potential recovery budget, and whether each detector family contributes unique evidence or merely duplicates signals elsewhere in the system. These questions matter because an overly aggressive configuration can create resilience loss through false isolation and unnecessary recovery, while an overly conservative configuration can detect attacks only after mission functionality has already degraded.

The design reuses the reliable-threshold attack magnitudes identified in Experiment 3 and holds the mission workload, attack timing, mission phase, and recovery policy fixed. Holding those variables constant isolates the effect of IDS configuration from the attack-intensity and mission-phase effects measured in Experiments 3 and 4. Each sub-study changes one configuration dimension while keeping the selected operating point from the other dimensions fixed once it has been established.

A **window-duration sweep** varies the signal window duration to measure the trade-off between fast response and noisy evidence. A **threshold sweep** varies detector thresholds to identify operating points that preserve high detection probability without excessive false positives.

A **detector ablation** disables individual mechanisms or runs with only subsets enabled to test whether each detector carries the dominant signal for its associated attack class. The dependent variables are detection probability, FPR/hr, and median MTTD; these are interpreted together rather than optimized independently because the useful operating point is the one that supports bounded response, not the one that maximizes a single metric.

6.8.2 Results

Experiment 5 reports three configuration sub-studies executed against Alcyone: a window-duration sweep (5A), a threshold sweep (5B), and a per-detector ablation (5C). The studies are reported separately because they answer distinct configuration questions and yield distinct operating-point recommendations.

5A: Window-Duration Sweep

The IDS evidence-window duration was varied from 1,000 to 60,000 ms with all other detectors held at default thresholds. Table 6.9 reports detection probability against three attack families, false-positive rate per nominal hour, and median MTTD.

Table 6.9 Experiment 5A: IDS evidence-window sweep. P_{detect} values are reported against command flood, sensor drop, and unauthorized command at the reliable thresholds from Experiment 3. FPR/hr is false-positive rate per nominal operating hour.

Window (ms)	P_{flood}	P_{sensor}	P_{auth}	FPR/hr	Median MTTD (s)
1,000	0.920	0.350	1.000	0.900	0.800
2,500	0.950	0.620	1.000	0.350	1.400
5,000	0.980	0.860	1.000	0.120	2.200
10,000	0.990	0.960	1.000	0.050	3.800
30,000	1.000	1.000	1.000	0.020	11.500
60,000	1.000	1.000	1.000	0.010	24.000

The selected operating point is the 10,000 ms window. It achieves $P_{\text{detect}} = 0.990$ for command flood, 0.960 for sensor drop, and 1.000 for unauthorized command, with FPR/hr

≤ 0.05 and median MTTD 3.8 s. Shorter windows reduce MTTD but raise FPR sharply: the 1,000 ms window exhibits 18$\times$ the false-positive rate of the 10,000 ms window for a 4.75$\times$ reduction in MTTD, while detecting only 35% of sensor-drop cases. Immediate reaction is operationally untenable for flight software: each false positive triggers an unnecessary recovery action that consumes the supervisor's bounded action budget.

Longer windows reduce FPR and eventually raise all three detection probabilities to 1.000, but at sharply increased MTTD. The 60,000 ms window achieves a 5$\times$ FPR reduction relative to the 10,000 ms window in exchange for 6.3$\times$ the MTTD. The 10-second selection is therefore neither minimum-FPR nor minimum-MTTD; it is the operating point that preserves high detection coverage while avoiding both excessive false positives and sluggish response for the Earth-observation mission profile.

5B: Threshold Sweep

The rate-detector threshold (`floodpivot`) was swept from 1 to 15, and the authority-detector threshold (`aclburst`) from 1 to 20. Table 6.10 reports the joint sweep.

The selected operating points are `floodpivot-thresh-5` and `aclburst-thresh-3`. At these thresholds, the rate detector achieves $P_{\text{detect}} = 0.95$ for command-flood attacks at the reliable-threshold rate (1,000 packets/s, from Experiment 3) with FPR/hr 0.05 and median MTTD 2.5 s. The authority detector achieves $P_{\text{detect}} = 1.000$ for unauthorized-command bursts of size 3 or greater with FPR/hr 0.04 and median MTTD 0.16 s.

The threshold sweep makes two architectural points concrete. First, the authority-detector reliable threshold (3 retries within the burst window) corresponds to the rejection budget allocated to legitimate operator typos and command-link transients; below threshold-3, detection probability remains 1.000 but FPR/hr rises above 0.10, which is operationally costly. Second, the rate detector's threshold-1 configuration produces $P_{\text{detect}} = 1.000$ but at FPR/hr 0.95—nearly one false alarm per nominal operating hour. The architecture supports either operating point; the selected one trades a small detection-probability margin for a 19$\times$ reduction in FPR.

Table 6.10 Experiment 5B: IDS detector threshold sweep for the rate detector (`floodpivot`) and the authority detector (`aclburst`). Selected operating points are `floodpivot=5` and `aclburst=3`.

Detector	Threshold	P_{detect}	FPR/hr	Median MTTD (s)
Rate (`floodpivot`)	1	1.000	0.950	0.800
Rate (`floodpivot`)	2	1.000	0.450	1.000
Rate (`floodpivot`)	3	0.990	0.180	1.600
Rate (`floodpivot`)	**5**	**0.950**	**0.050**	**2.500**
Rate (`floodpivot`)	8	0.850	0.020	4.000
Rate (`floodpivot`)	15	0.550	0.005	7.500
Authority (`aclburst`)	1	1.000	0.300	0.100
Authority (`aclburst`)	2	1.000	0.100	0.140
Authority (`aclburst`)	**3**	**1.000**	**0.040**	**0.160**
Authority (`aclburst`)	5	1.000	0.010	0.180
Authority (`aclburst`)	10	0.850	0.002	0.300
Authority (`aclburst`)	20	0.400	0.000	0.550

5C: Per-Detector Ablation

The ablation study held the rate-detector and authority-detector thresholds at their selected operating points (5B) and the evidence window at 10,000 ms (5A). Each attack family was then run with four detector configurations: full IDS, IDS off, all detectors except the relevant one (`minus-target`), and only the relevant detector (`only-target`). Table 6.11 reports the result.

The ablation establishes that no single detector family covers the attack space. With IDS disabled, all three attack classes produced $P_{\text{detect}} = 0.000$. With the targeted detector removed, detection collapsed: $P_{\text{detect}} = 0.300$ for command flood (residual signal in queue-pressure telemetry), 0.200 for unauthorized command (some bursts crossed the authority boundary indirectly through mode-transition validation), and 0.250 for sensor drop (residual signal in attitude-control divergence). With only the targeted detector enabled, detection recovered to $P_{\text{detect}} \geq 0.95$, confirming that each detector family carries the dominant signal for its attack class.

The full-detector configuration outperforms the most aggressive single-detector configuration

Table 6.11 Experiment 5C: per-detector ablation. Each attack family (f1 command flood, f2 unauthorized command, f4 sensor drop) is run under four configurations.

Attack	Configuration	P_{detect}	FPR/hr	Median MTTD (s)
f1 (cmd flood)	full	0.980	0.050	3.800
f1 (cmd flood)	ids-off	0.000	0.000	—
f1 (cmd flood)	minus-target	0.300	0.020	8.000
f1 (cmd flood)	only-target	0.950	0.040	2.500
f2 (unauthorized)	full	0.980	0.050	0.160
f2 (unauthorized)	ids-off	0.000	0.000	—
f2 (unauthorized)	minus-target	0.200	0.040	—
f2 (unauthorized)	only-target	1.000	0.010	0.200
f4 (sensor drop)	full	0.980	0.050	8.300
f4 (sensor drop)	ids-off	0.000	0.000	—
f4 (sensor drop)	minus-target	0.250	0.040	—
f4 (sensor drop)	only-target	0.950	0.010	8.500

on FPR while matching it on P_{detect}. The ablation argues against single-point IDS designs at the mechanism level: each attack family leaves a different evidence signature, so detector mix must be chosen against observed coverage, not assumed completeness. The integration of those families into a coherent evidence stream is itself an architectural property the ablation tests.

The configuration sub-studies together favor a defensible operating point rather than the most aggressive one. For flight software, a configuration that detects slightly later but avoids repeated false isolation is preferable to a configuration that reacts immediately to benign bursts. The IDS contributes to resilience when its thresholds, windows, and detector mix are chosen against a mission-specific risk model and verified against nominal workloads.

The selected operating point—10,000 ms evidence window, `floodpivot=5`, `aclburst=3`, full detector mix—is the configuration used in Experiment 1. Experiment 5 therefore validates retrospectively that the comparative result of Experiment 1 rested on a defensible configuration rather than on hidden defaults: the window-duration and threshold sweeps locate Experiment 1's operating point in a region where detection probability remains ≥ 0.95 for command-flood and sensor-drop attacks at FPR/hr ≤ 0.05, and the ablation confirms that no detector family in the

Experiment 1 configuration is doing redundant work.

6.9 Threats to Validity

6.9.1 Internal Validity

The primary internal validity concern is testbed circularity. Taurus, the adversarial testbed, was produced within the same research effort that produced Alcyone itself. The adversarial scenarios derive from the architecture's engineering artifacts and execute against the flight software as a black box, providing behavioral coverage independent of the implementation's internal structure. That independence holds regardless of how the test harness was built. However, the threat model, the scenario catalog, and the Taurus simulator implementations share a common authorship context with the system under test. Truly independent validation would require a red team operating against a specification produced independently of the implementation workflow.

A related concern is single-author test bias. Unit tests were written alongside the implementation, verifying internal consistency with the developer's interpretation of the specification rather than correctness against independent ground truth. This limitation applies to any single-author development process. The interoperability tests against NASA CryptoLib (Section 5.6.1) partially mitigate the concern for protocol implementations by providing an external reference, but no equivalent external oracle exists for application-layer behavior.

The NOS3 SWIL environment introduces simulation fidelity as a confound. Hardware simulators approximate but do not replicate physical device behavior: timing jitter, bus contention, and analog noise are absent or simplified. Architectural differences that manifest under simulation may be amplified or attenuated relative to flight hardware. No hardware-in-the-loop validation is included in this evaluation, so the adversarial results characterize controlled SWIL behavior rather than flight-hardware behavior. The seL4 substrate that hosts Alcyone in the campaign is exercised under QEMU emulation rather than on flight-class hardware; Determin-

istic QEMU scheduling and the absence of physical interrupt-timing variation may sharpen isolation-boundary properties relative to bare-metal execution.

6.9.2 External Validity

The evaluation is scoped to a single mission profile (an Earth-observation small satellite in LEO), a single implementation language (Rust), and a specific architectural approach centered on service isolation with seL4 protection domains as the deployment substrate, exercised under QEMU emulation in the SWIL testbed. The comparison tests whether the *class of design choices* Alcyone represents—process isolation, per-command authorization, compile-time memory safety—produces measurably different resilience outcomes than the conventional class represented by cFS, not whether Alcyone is universally superior. Missions with different threat profiles or constraint environments may weight these properties differently. The cFS configuration adopts the default component set without mission-specific security additions, since no standardized cyber-hardening baseline exists for the framework.

6.9.3 Construct Validity

The resilience ratio R integrates detection time, recovery time, and functionality preservation into a single scalar. Any composite metric involves weighting choices that privilege certain resilience properties over others. The metric definitions and weighting rationale are documented in Section 6.3; readers who weight detection speed differently from recovery completeness may interpret the same raw data differently. Experiment 2 addresses this construct-validity risk by reporting level-specific traces and attenuation gaps rather than relying only on an aggregate scalar.

The SPARTA threat catalog provides the scenario basis. SPARTA is the most complete publicly available catalog of space system attack patterns, but it is not exhaustive. Attack techniques that fall outside SPARTA's current scope—supply chain compromise during integration, for example—are represented only indirectly through adaptive kill chains rather than as first-class

scenarios. The scenario catalog tests the architectural properties identified as consequential in earlier chapters; it does not test all properties that might matter.

6.9.4 Experimental Design Trade-offs

Experiment 1 uses a paired single-trial design across thirteen scenarios. Three properties of the campaign bound what that design supports.

First, the substrate is deterministic. NOS3 orchestrates flight-software execution under fixed seeds, scheduling, and dynamics initial conditions; the QEMU-emulated seL4 substrate executes Alcyone tasks on a deterministic scheduler; SPARTA-mapped scenario stimuli are scripted from time-tagged Taurus campaign definitions. Trial-to-trial variance under this configuration is bounded by sources whose magnitude does not approach the per-scenario differences reported in Table 6.4—differences typically of an order of magnitude in MTTD and a factor of two or more in resilience ratio across architectures.

Second, the comparative claim is structural rather than distributional. Experiment 1 reports per-scenario direction and magnitude of the Alcyone-versus-cFS difference: 12 of 13 scenarios produced a qualifying onboard detection in Alcyone and 3 of 13 in cFS, with the difference traceable to specific architectural enforcement surfaces (per-command authorization, typed bus policy, supervisor-mediated recovery). A 12-of-13 outcome on thirteen paired scenarios is not strengthened arbitrarily by adding within-cell replication; what would strengthen it is an enlarged or differently sampled scenario suite.

Third, replicated-trial designs appear in the campaign where the dependent variables require them. Experiments 3, 4, and 5 use three trials per cell because detection probability, FPR/hr, and median MTTD require within-cell variance estimates. Experiment 1's dependent variables are categorical (detection or no detection, accepted adversary action or not) or scenario-aggregated; replicating the same scenario does not change the categorical outcome under deterministic execution.

Experiment 1's evidence is therefore a function of the scenario suite rather than of within-

scenario sampling. A different mission profile, a different attack-content set, or a different cFS configuration would produce different scorecard cells; Section 6.9.2 bounds those generalization questions.

6.10 Summary

The five experiments substantiate the architectural claims of Chapter 5 along the resilience-metric framework defined in Section 6.3.

Experiment 1 shows the comparative separation: Alcyone produced a qualifying onboard detection in 12 of 13 scenarios versus three for cFS (3/13), with lower mean MTTD, higher mean resilience ratio, and smaller mean blast radius. The separation is largest at authority boundaries, where per-command authorization prevents unauthorized-command and command-replay cases from becoming accepted system actions.

Experiment 2 explains where Alcyone preserves mission utility. Across four representative attack classes, mission-level resilience ratios exceed component-level ratios by $+0.012$ to $+0.070$, showing that service, bus, supervisor, and mission-control boundaries damp local disturbance before it becomes mission-level consequence. Experiment 4 adds the timing context: detection latency is stable across the tested phases, but mission consequence depends on whether the attack occurs during imaging, coast, or downlink.

Experiments 3 and 5 bound the IDS operating envelope. Intensity sweeps identify reliable thresholds for authority-boundary, command-flood, sensor-drop, and bus-spoof attacks; configuration sweeps show how windows and thresholds trade detection probability, latency, and false-positive cost. The ablation demonstrates that no single detector family is sufficient: removing the targeted detector for an attack class collapses P_{detect} to between 0.20 and 0.30.

Architectural interpretation of the measured results is carried forward to Chapter 7.

Chapter 7
Discussion

Chapter Abstract. The contributions of Chapters 3–6 have implications for space cybersecurity practice, standardization, and broader mission classes. The four research questions of Chapter 1 have direct answers in the empirical results and derives concrete recommendations for practitioners, program managers, and standards bodies. It also examines how the flight-software arguments extend to cislunar and proliferated mission classes, where existing operational assumptions about response time and connectivity no longer hold.

7.1 Answers to the Research Questions

7.1.1 RQ1: Threat Landscape and Salient Attack Surfaces

Five structural pressure points dominate contemporary flight software stacks: (i) OSAL complexity that obscures the platform interface, (ii) single-address-space execution that prevents containment, (iii) debug facilities that become post-exploitation primitives, (iv) permissive BSP defaults that expand reachable surface, and (v) memory unsafety that converts ordinary interfaces into code-execution pathways. The bottom-up decomposition of cFS on RTEMS in Chapter 3 produced these findings, and the decomposition itself is the methodological answer to RQ1 [13].

The key finding is structural: shared address space, implicit internal trust, and limited onboard authority enforcement are not isolated bugs to be patched individually; they are architectural conditions that make attack effects compound. The single-address-space model means a memory corruption in one component can reach any other, and the absence of message-level authentication on the Software Bus means a compromised component can impersonate any other. Bottom-up decomposition reveals risks that top-down threat modeling can miss: bound-

aries and controls at the RTOS, BSP, and abstraction-layer interfaces have disproportionate impact because they mediate all higher-level behavior.

These findings directly shaped the architecture and evaluation. The single-address-space risk motivated Alcyone's process-isolated service model (Chapter 5). The OSAL complexity findings informed the decision to eliminate C abstraction layers entirely. The permissive-default problem drove the requirement for deny-by-default access control on all inter-service communication. Without the bottom-up decomposition, these architectural decisions would have been intuitions rather than threat-derived design responses.

RQ1 identifies which weaknesses have architectural leverage rather than merely cataloging them. Attack-surface analysis is not an endpoint or a vulnerability inventory; it is the mechanism that determines where resilience boundaries must exist if later requirements and design claims are to be credible.

7.1.2 RQ2: Minimum and Testable Requirements

A secure-by-component workflow generates testable requirement statements for flight-software components, with explicit traceability from threats to enforcement points [15]. That workflow rests on a coauthored minimum-requirements foundation that scopes "minimum" to a single mission-level failure mode—permanent loss of control—and performs fault-tree-informed coverage analysis to identify the secure-by-design principles each mission segment requires [14]. Together the two methodologies answer RQ2 by specifying both what makes a cybersecurity requirement worth writing down and how to derive a minimum set scoped to mission-critical control and availability.

The threat-informed workflow in Chapter 4 shows how this works in practice: mission-specific requirements are derived from adversary assumptions and scoped to component-level enforcement points, and existing quality assurance processes (coding guidelines, review gates, tool integration) serve as natural insertion points for their implementation [9].

Methodologically, the requirements work links threat analysis, architecture, and evaluation.

Requirements derived from a threat-informed workflow inherit the mission objective and adversary assumptions that scoped them, and they remain traceable to threats and verification strategy. Chapter 5 instantiates these requirements as enforcement points in Alcyone's architecture, and Chapter 6 uses the same threat-to-requirement mapping to design evaluation scenarios. The chain from threat technique to requirement to enforcement surface to test scenario makes resilience claims auditable rather than aspirational. A requirement worth writing down, in this framework, is one that is traceable to a mission-relevant threat model, localizable to an enforcement surface, and verifiable through an explicit test or proof obligation.

7.1.3 RQ3: Architecture That Embeds Resilience by Construction

Alcyone (Chapter 5) is the architectural answer to RQ3: a secure-by-component decomposition in which requirements are instantiated at interface boundaries and where isolation and integrity checks are treated as first-class architectural elements rather than optional extensions [16, 8].

The principles underlying Alcyone (isolation, secure input handling, memory safety, and formal verification of critical invariants) were identified through the attack surface analysis in Chapter 3 and the requirements derivation in Chapter 4. Alcyone's contribution is demonstrating that these principles can be composed into a single architecture without sacrificing the modularity and testability that flight software demands [9].

Alcyone makes an architectural hypothesis testable: structural isolation, privilege constraints, and integrity validation are expected to make compromise more containable and recovery behavior more predictable. Chapter 6 defines the benchmark and measurement strategy used to test that hypothesis empirically.

Each structural decision trades against the flexibility and performance priorities of conventional flight software. Three matter most. Centralizing recovery authority in a single Supervisor service makes mode reachability analyzable in one place, but creates a critical service that must be protected by a hardware watchdog, deterministic state recovery, and a simple mode state machine. Copying inter-service messages eliminates a class of data-injection attacks, but

imposes a per-message copy cost that is acceptable only because Alcyone's payload size and service count are bounded. Abstracting platform differences through Rust trait generics provides compile-time enforcement of privilege constraints, but ties the architecture to a language ecosystem still maturing toward safety-critical qualification. The architecture decision records trace these choices to specific threat-model requirements and document their consequences for performance, analyzability, and deployment flexibility [16].

The resulting claim is that resilience can move from policy intent into the architecture itself. Once trust boundaries, authority limits, and recovery actions are structural properties, they become reviewable engineering decisions rather than informal expectations imposed on developers and operators.

7.1.4 RQ4: Evaluation and Measurement

Chapter 6 answers RQ4 through an evaluation framework: a paired architectural benchmark, Taurus scenario automation, repeated adversarial campaigns, timing metrics (MTTD, MTTR), and a resilience ratio R adapted from Weisman et al.'s functionality-based protocol [109]. These elements address the anticipate/withstand/recover/adapt framing of NIST SP 800-160 Vol. 2 [8] while remaining measurable in SWIL environments and sensitive to architectural differences.

The framework makes the evidence behind a resilience claim explicit. Detection definitions, recovery markers, functionality models, and event schemas are treated as part of the claim rather than as bookkeeping. Taurus scenario automation and SPARTA-mapped campaigns make that evidence repeatable, so mission teams can test whether adversaries can exploit flight software functionality and whether architectural choices materially change the outcome [9].

The results identify where Alcyone creates resilience. Its advantage is concentrated at enforcement surfaces: command authority, typed bus policy, explicit health evidence, and supervisor-controlled recovery. The 12-of-13 versus 3-of-13 detection split in Experiment 1 matters because it reflects a shift from forensic reconstruction to onboard evidence generation; the larger operational effect is that anomalous commands and bus messages become explicit

rejections, containment actions, or bounded recovery decisions before they spread through the mission state.

The multi-level resilience metric supplies the mission-level interpretation missing from detection tables. Positive attenuation gaps mean that local disturbance is damped before it becomes mission consequence. The sweep, phase-timing, and ablation studies then bound the claim: low-intensity attacks can fall below selected operating points, operational consequence depends on mission phase, and IDS performance is an engineering operating point rather than a fixed property of the architecture.

For spacecraft mission programs and acquisition teams, evaluation becomes a set of concrete questions: which enforcement surface produced evidence, whether local impairment reached mission outputs, which mission phase made the effect costly, and what false-positive cost the response policy can tolerate. This shifts evaluation from demonstration to engineering trade study while keeping the limits of SWIL evidence visible.

7.2 Technical Implications for Flight Software Architecture

The technical implication of the evaluation is not that any single mechanism supplies resilience on its own. Alcyone's advantage appears where threat-derived requirements become enforcement surfaces that change how compromise propagates, how evidence is produced, and how recovery authority is constrained. Architecture, implementation language, formal checks, and SWIL measurement matter because they are tied to those surfaces.

7.2.1 Enforcement Surfaces and Interface Policy

Resilience at the flight-software layer begins with authority boundaries that an exploit cannot silently cross. Experiment 1 showed that Alcyone's measured advantage appeared where command validation, typed bus policy, service isolation, and supervisor-mediated recovery generated onboard evidence and bounded propagation. Experiment 2 showed that those local

boundaries damped disturbance before it became mission-level consequence. Isolation is useful when it changes the attack path, not merely when it appears as a design feature.

The highest-risk assumptions often live at interfaces rather than inside individual components. Bus protocols, shared memory regions, inter-processor links, debug channels, recovery commands, and maintenance paths can all become privilege-escalation surfaces if they sit outside the same policy model as nominal command and telemetry. The RTEMS shell illustrates the risk: a development convenience can expose file-system access, system queries, memory editing, and dynamic-loader management through serial or network access in a single-address-space RTOS [13]. A resilient architecture treats those auxiliary interfaces as part of the security argument, not as integration details.

Alcyone's separated detection and recovery authority illustrates this boundary discipline. Detection services observe and recommend, but only the Supervisor can act, and its actions are limited by typed bus boundaries, cooldowns, and a closed recovery set. That separation makes recovery behavior reviewable as policy rather than an emergent effect of application code.

7.2.2 Evidence Integrity and Measurement Credibility

Substantiated integrity is the basis for trusting the evidence that a spacecraft uses to make recovery decisions. A detector can report an anomaly, a supervisor can issue a recovery action, and telemetry can show resumed function, but those artifacts support a resilience claim only if the command stream, telemetry path, event log, software image, and runtime state are protected from silent alteration. Integrity is therefore part of the measurement system itself, not a separate control applied after the fact.

This matters because the evaluation metrics are built from evidence produced during each scenario. MTTD depends on when an attack becomes observable; MTTR depends on recovery markers and resumed service behavior; blast radius and functionality scores depend on telemetry and state traces. If an attacker can forge commands, suppress logs, rewrite telemetry, or

substitute software, the same run could appear successful while the underlying mission state remains compromised. The measured result would then describe an integrity failure in the evidence path rather than the resilience of the architecture.

For flight software, credible resilience evidence requires protected evidence paths: authenticated commands, integrity-protected telemetry, software provenance checks, attestation where available, and tamper-evident event records. Alcyone makes those paths architectural by tying observations and recovery decisions to typed bus messages, supervisor-controlled authority, and traceable enforcement surfaces. These mechanisms do not prove that every runtime state is correct, but they make recovery claims auditable: the program can identify what was observed, what authority acted, and which evidence supports the conclusion that mission function was preserved.

7.2.3 Implementation Assurance and Residual Engineering Limits

The implementation-level implication is narrower than a general argument for Rust or memory-safe languages. Memory safety, type invariants, static analysis, bounded model checking, and deductive proofs contribute to resilience when they protect the authority, routing, parsing, and recovery boundaries that the architecture depends on. A memory-safe service model prevents ordinary input-handling defects from bypassing those boundaries from below; typed messages and proof harnesses make selected interface assumptions mechanical rather than advisory.

Alcyone's verification experience also shows why implementation assurance must remain tied to live code. Kani proof harnesses found two latent defects in a codebase with high test coverage: an unchecked arithmetic wraparound in FDIR self-assessment and overlapping opcode constants that made command disambiguation fragile. Those defects matter here because they affected recovery evidence and mode authority, not because they are unusual programming mistakes. Coverage metrics do not substitute for property-oriented checks over the state space that matters to resilience.

The Verus proof-drift findings sharpen the same point. Two proof modules still verified

after the implementation they modeled had changed, demonstrating that a proof can remain internally consistent while losing correspondence with production code. This is the strongest practical argument for co-located proofs, CI execution, and explicit traceability from requirement to invariant to implementation. The remaining boundary is timing: Alcyone's evaluation does not establish hard real-time guarantees, and Rust does not by itself provide worst-case execution time evidence. The dissertation's implementation claim is therefore containment, authority, and invariant-based correctness; hard timing assurance remains a separate engineering obligation.

7.3 Operational, Program, and Policy Implications

The operational and policy implication is that resilience evidence has to appear at the same gates where programs already make design, integration, acquisition, verification, and operational decisions. A late security review cannot substitute for attack-surface analysis, traceable requirements, architectural enforcement, adversarial evaluation, and lifecycle governance.

7.3.1 Budgeted Autonomous Response

Autonomous response is an operational policy problem as much as a technical mechanism. If a spacecraft cannot rely on immediate ground response, it needs onboard evidence, containment decisions, and recovery actions that are bounded before flight. The evaluation makes that trade visible through MTTD, MTTR, blast radius, false-positive rate, and the resilience ratio R.

The Supervisor provides one concrete model: it evaluates detection events against thresholds and response budgets, issues only actions from a closed set, and escalates when a budget is exhausted. Experiment 5B shows why this matters. Lower thresholds make the system more reactive but consume recovery budget through false alarms, while higher thresholds preserve budget but can delay evidence. Experiment 4 adds that the same detected event has different operational consequence depending on mission phase. A defensible response policy therefore ties detector operating points to mission phase, contact opportunities, tolerance for nuisance

recoveries, and degraded-mode rules.

7.3.2 Reuse, Requirements, and Risk as Program Gates

Mission software reuse makes attack-surface review part of configuration and integration, not only a code review of mission applications. The cFS/RTEMS analysis shows why: inherited defaults, abstraction layers, debug interfaces, and operating-system facilities can create mission-relevant attack paths before new mission code is written [13]. Programs that reuse flight software need a gate that asks what authority, data flow, and recovery assumptions the reused stack brings with it.

That gate should feed directly into requirements and risk analysis. Requirements that cannot be verified in SWIL, formal analysis, test, or review are weak evidence for resilience claims; risk findings that are never attached to enforcement surfaces remain analytical observations. The secure-by-component workflow and recent space cyber risk methodology are complementary for this reason: risk analysis identifies which control and data flows matter most, while architectural enforcement prevents those flows from becoming unbounded compromise paths [15, 53].

7.3.3 Acquisition, Standards, and Lifecycle Governance

Acquisition and standards efforts should emphasize secure-by-design building blocks that map cleanly to enforcement evidence: segmentation and isolation, integrity validation, privilege constraints, memory-safe implementation expectations, and adversarial test artifacts [42]. Threat frameworks such as SPARTA provide a common adversary vocabulary, while standards such as IEEE P3349 can make the expected evidence portable across programs.

The same logic applies across the mission lifecycle. During concept definition and design, cybersecurity requirements belong alongside functional and safety requirements. During implementation, integration, and V&V, memory-safe languages, static analysis, formal checks, and adversarial SWIL testing belong in the normal engineering workflow rather than in a separate security lane. During operations, sustainment, and decommission, programs need monitored

update paths, preserved trust boundaries, onboard autonomy, and end-of-life plans for disabling command interfaces where feasible. The governance point is organizational: cybersecurity expertise must be embedded in systems engineering decisions from the outset, not consulted only during late-stage review.

7.4 Boundary of the Dissertation Claim

Chapter 6 catalogs the experiment-level validity threats. The dissertation-level question is different: what claim does the integrated evidence chain support? The answer is bounded but substantive. Threat analysis can be translated into testable requirements; those requirements can be embodied as flight-software enforcement surfaces; and the resulting architecture can be evaluated through observable resilience behavior in a controlled SWIL campaign. It does not claim flight qualification, universal architectural superiority, complete mission-system security, or a final standards baseline for all mission classes.

7.4.1 What the Evidence Establishes

The strongest claim is methodological. The threat-to-requirement-to-architecture trace is explicit and reviewable, and the evaluation closes that trace by attaching detection, recovery, functionality, containment, and evidence-integrity outcomes to the same adversarial scenarios. Cyber resilience is therefore an architectural property of flight software rather than a late operational add-on.

Empirically, the claim is narrower. Taurus shows that, under the campaign conditions, Alcyone's enforcement surfaces changed observable resilience behavior relative to the cFS baseline. The result is strongest where the scenario directly exercises command authority, bus policy, supervisor-mediated recovery, or telemetry evidence—the places where a threat-derived requirement points to an architectural mechanism and then to a measured outcome.

7.4.2 What the Evidence Does Not Establish

The evidence does not establish that Alcyone is flight-qualified, that Rust is sufficient for spacecraft cybersecurity, or that any single architecture is universally superior. It also does not establish hard real-time timing assurance, hardware-fidelity behavior on flight processors, or complete end-to-end mission security. Those claims would require independent adversarial assessment, representative hardware validation, qualification evidence, and analysis of ground, supply-chain, cryptographic, and operational controls beyond the flight-software layer.

The resilience metrics should be read the same way. They make architecture-specific behavior observable, but they are not absolute mission-assurance scores. The resilience ratio adopted in Chapter 6 comes from Weisman et al.'s functionality-based measurement protocol for cyber-resilient systems [109] and is adapted here to flight software. The functionality decomposition, recovery markers, and weighting scheme support comparison within this campaign; applying them as certification or acquisition criteria would require separate validation.

7.4.3 Generalization Boundary

The scope is flight software executing on the vehicle computer and threats that couple to command, telemetry, and on-board control behavior. Ground-segment compromise, supply-chain threats, insider threats, and link-layer cryptography are addressed only as boundary conditions. This scope is deliberate because it isolates high-leverage on-board design decisions, but it also means a complete mission security posture must account for segments and authorities outside the vehicle computer.

Generalization therefore requires revalidation rather than direct transfer. A different mission profile, contact cadence, processor, RTOS, payload, operator procedure, or adversary model could change which requirements matter most and how resilience metrics should be weighted. Future work broadens that empirical boundary: additional runtime baselines, representative hardware, stronger proof-to-code continuity, multi-spacecraft testbeds, and governance models for missions that span organizations.

7.5 Future Work

The immediate next step is to test whether the evidence chain — threat analysis, derived requirements, architectural enforcement, and adversarial evaluation — holds across broader platforms, stronger adversarial conditions, more realistic hardware constraints, and multi-actor mission settings. Three directions are most consequential for the central evidence chain: comparative runtime evaluation across space RTOSs (Section 7.5.1), hardware-rooted trust on multi-processor flight platforms (Section 7.5.2), and continuity between requirements and formal evidence (Section 7.5.3). The remaining directions extend the framework to broader operational settings: detection and response mechanisms within flight-processor constraints, resource-constrained security, distributed-spacecraft resilience, higher-fidelity testbeds, cislunar and proliferated mission class extensions, and multi-stakeholder governance.

7.5.1 Space-Specific RTOS Evaluation and Replacement

The attack surface analysis in Chapter 3 characterizes vulnerabilities in one RTOS (RTEMS) paired with one framework (cFS). A natural extension is a comparative evaluation across the RTOS options available for space missions (RTEMS, VxWorks, FreeRTOS, Zephyr, seL4) using a common framework that scores memory protection, privilege separation, secure boot support, and susceptibility to the vulnerability classes identified in the attack surface catalog [13]. This comparison would test whether the weakness classes found in RTEMS/cFS generalize across flight software platforms, and whether different runtime substrates materially change the resilience envelope measured in Chapter 6.

7.5.2 Hardware-Rooted Trust and Multi-Processor Architectures

Alcyone's trust model combines language-level isolation, bus-enforced topic access control, and, in its intended deployment, seL4-backed protection domains and scheduling contexts. A hardware-focused extension would test whether those boundaries can be strengthened

with hardware-rooted guarantees rather than only software-enforced policy. One direction is an *immutable core* architecture in which critical infrastructure services (the message bus, supervisor, and health monitor) reside in read-only regions enforced by a memory protection unit or memory management unit. Another is a multi-processor deployment, as envisioned for heterogeneous platforms such as NASA's High-Performance Spaceflight Computing (HPSC), with an authenticated inter-processor bus bridge that enforces topic access control at processor boundaries. These extensions would test whether the architectural decisions described here (stateless critical services, deterministic recovery from persistent state, copy-based message passing) remain stable when hardware roots of trust and inter-processor communication are added [16].

7.5.3 Formal Verification of Flight Software Components

The open problem is evidence continuity, not whether formal methods are useful in principle. Chapter 5 demonstrates CI-integrated Kani and Verus proofs for selected Rust components, while Chapter 4 introduces formal methods as part of a requirements derivation workflow. The next step is to connect those layers: requirements-level properties, implementation proofs, generated binaries, runtime configuration, hardware interface layers, and communication stacks. The goal is a continuous assurance chain that reduces the proof-drift problem identified in Section 7.2.3 and evaluates whether component-local proofs remain meaningful once the software is linked into a flight stack.

7.5.4 On-Board Anomaly Detection and Autonomous Response

The evaluation framework in Chapter 6 measures detection and recovery as MTTD and MTTR, but the detection mechanism itself, threshold-based telemetry monitoring, is deliberately simple. A natural extension is to evaluate lightweight machine-learning models (e.g., EWMA/CUSUM detectors, one-class classifiers trained on nominal behavioral baselines) that can run within the resource constraints of a flight processor. The threshold-detector operating points reported

in Section 6.8.2—$P_{\text{detect}} \geq 0.95$ at FPR/hr ≤ 0.05 across three attack families—define the floor that any proposed ML detector should meet or exceed before the additional verification burden is justified. The research question is whether such models can reduce MTTD for subtle attacks (sensor bias injection, slow command-rate manipulation) without raising the false-positive rate that triggers unnecessary safe-mode transitions. Reinforcement learning for response policy selection (isolate vs. reconfigure vs. alert ground) is a further direction, though it raises verification challenges for safety-critical autonomy.

Two additional challenges define the research boundary. First, spacecraft operate in noisy environments where single-event upsets, sensor glitches, and thermal transients can resemble adversary-induced deviations. Differentiating cyber attack from random fault, especially when an adversary may deliberately mimic natural failure modes, requires uncertainty estimates that operators and autonomous response systems can use. Second, the detection model becomes part of the attack surface: adversarial inputs crafted to produce incorrect classifications could cause a detection system to miss real attacks or trigger false alarms that degrade mission performance.

7.5.5 Resource-Constrained Security

Alcyone's security mechanisms assume enough computational headroom for authenticated messaging, behavioral baseline tracking, and anomaly detection. Small satellites and CubeSats may not have that margin: radiation-hardened processors often lag commercial processors in performance, and power budgets can limit continuous cryptographic operations or inference. A resource-focused extension would quantify the overhead of Alcyone's security extensions on representative flight-class processors and determine which mechanisms can be scaled down for resource-limited platforms while preserving their core security properties.

7.5.6 Compromised-Node Resilience for Distributed Space Systems

The preceding analysis addresses single-vehicle architectures, but future space operations will increasingly involve satellite constellations and multi-vehicle formations where the cyber compromise of one node can affect the collective. The relevant Byzantine behavior is not ordinary component failure; it is a compromised spacecraft, processor, or relay that continues participating while sending plausible but inconsistent, delayed, or adversary-controlled messages. Such a node could falsify detection alerts, corrupt tasking or custody-transfer messages, suppress relay data, or induce the formation to allocate mission functions around a malicious state estimate.

Traditional N-modular redundancy with voting can isolate a unit producing incorrect outputs due to a random fault, but identical redundant units running the same software are vulnerable to common-mode cyber attacks. The open question is whether Byzantine-resilient protocols can support cyber-specific collective defenses—detection-alert quorum, compromised-member isolation, authenticated relay disagreement, and mission-function redistribution—under spacecraft latency, bandwidth, and autonomy constraints. Multi-spacecraft testbeds would provide a foundation for evaluating whether those defenses preserve mission function when one member is no longer trustworthy.

7.5.7 Testbed Extensions for Multi-Spacecraft and Hardware-in-the-Loop Validation

The SWIL testbed described in Chapter 6 demonstrates that adversarial evaluation of flight software architectures is feasible in simulation. Two extensions would increase the fidelity and scope of this approach. First, hardware-in-the-loop configurations, running flight software on representative processors with real bus interfaces, would validate that timing-dependent security properties (watchdog response, anomaly detection latency) hold outside the simulator's idealized scheduling. Second, multi-spacecraft topologies would enable evaluation of inter-vehicle trust boundaries, relay authentication, and cascading-failure scenarios relevant to constellation architectures [13].

7.5.8 Cislunar and Proliferated Mission Class Extensions

The dissertation's evidence chain is scoped to LEO operations with regular ground contact. Cislunar and deep-space missions amplify communication delay, store-and-forward dependency, and multi-hop trust assumptions—conditions that make immediate ground intervention less reliable and increase the operational value of onboard authority boundaries, trustworthy evidence, and bounded recovery [12]. The research question is whether the same evidence chain—threat-derived requirements, enforceable software boundaries, measurable resilience behavior, and standards-compatible evidence—can be preserved under longer contact gaps and authority distributed across multiple operators. Concrete extensions include detection windows and recovery budgets calibrated to delay-tolerant operations rather than to LEO contact cadence, attestation and provenance mechanisms that survive multi-hop relay, and evaluation scenarios that exercise compromise of intermediate relays rather than only the vehicle under test. Emerging standardization work such as IEEE P3349 [42] provides a venue for making that evidence portable across organizations governed by different regulatory regimes.

7.5.9 Multi-Stakeholder Trust and Governance

Standards adoption is only one part of multi-stakeholder resilience; shared infrastructure also requires security mechanisms and governance models that work across organizations. This direction evaluates zero-trust architectures for space systems, including dynamic access control, continuous authentication, and federated identity management for multi-stakeholder missions. It also exercises tabletop scenarios for incident attribution, liability allocation, and coordinated response. Existing national directives such as SPD-5 [26] establish cybersecurity principles for domestic operators, but they do not resolve cross-jurisdictional authority or obligations among operators governed by different regulatory regimes.

Chapter 8
Conclusion

This dissertation began from a practical problem: spacecraft flight software is becoming more exposed to adversarial conditions, while much of its engineering tradition still treats cybersecurity as separate from reliability, safety, and mission assurance. That separation is no longer adequate. A compromised command path, permissive debug interface, or memory-corruption defect can affect the same control loops, telemetry flows, and recovery paths that keep a spacecraft mission-capable.

Cyber resilience in flight software rests on a chain of engineering evidence. Threat analysis identifies where compromise can gain leverage; requirements translate those observations into testable obligations; architecture embeds those obligations at enforcement surfaces; and evaluation measures whether those surfaces preserve mission function under attack. Resilience is therefore a property that can be designed, traced, and tested rather than a slogan or post-hoc control.

Viewed across the preceding chapters, three conclusions follow. First, the most consequential weaknesses in contemporary flight software are architectural, not merely implementation-local: shared address spaces, permissive defaults, and poorly bounded interfaces create systemic opportunities for compromise. Second, resilience becomes tractable when requirements are derived from explicit threats and attached to concrete enforcement surfaces rather than copied from generic checklists. Third, architectural decisions such as isolation, privilege restriction, message authentication, and memory-safe implementation can be justified as engineering responses to those threats and carried forward into verification and measurement. Together these constitute an end-to-end way of reasoning about cyber resilience in flight software rather than any single mechanism.

8.1 Summary of Contributions

8.1.1 Research Framing and Problem Scope

A widening gap exists between reliability engineering and cyber resilience as spacecraft become more software-driven [9]. The gap is architectural and evidentiary: conventional assurance can show correct behavior under expected conditions, but it does not establish whether the same system can anticipate adversarial behavior, withstand compromise, recover mission functions, or adapt its defenses over time [8]. Cyber resilience therefore belongs in flight-software design, not in an external review after the architecture is fixed.

Flight software is treated as the on-board cyber-physical control layer that mediates command handling, telemetry, fault response, mode control, and mission execution. Ground systems, payload software, communications security, and supply-chain concerns remain relevant to spacecraft security, but adversarial inputs reach spacecraft behavior most directly through flight software. This scope makes the research program tractable: threat analysis can identify exploitable interfaces, requirements can be tied to specific enforcement points, architecture can implement those points, and experiments can measure whether mission functionality is preserved under attack.

Cyber resilience is therefore a property to be supported by threat-scoped evidence, not assumed from reliability practice or generic secure-design advice.

8.1.2 Threat Landscape and Attack Surfaces

Chapter 3 supplies that missing analysis for a representative open-source stack: NASA's cFS running on RTEMS [13]. The bottom-up decomposition reveals inherited and configuration-dependent attack surfaces—the RTEMS shell, file system configuration, dynamic loader, and OSAL wrapping logic—that top-down threat modeling alone does not capture [13]. Shared-address-space execution and abstraction-layer indirection compound the problem: additional code layers expand the surface, while the absence of hardware-enforced isolation means a

single compromise can affect the entire system.

Memory safety supports those findings as a structural resilience lever. If the implementation language permits memory corruption, architectural isolation cannot fully contain compromise. Vander Stoep and Rebert [61] report that progressive Rust adoption in Android reduced memory-safety vulnerabilities to a small fraction of new defects, and similar patterns hold in Chromium [60, 62]; in flight software, the same defect class drives the attack-surface results in Chapter 3.

8.1.3 Minimum and Testable Requirements

Chapter 4 develops a secure-by-component workflow that translates threat-informed analysis into component-local, testable “shall” statements. Applied to the C&DH subsystem, the workflow shows that decomposing critical functions and analyzing their attack surfaces produces requirements that are both traceable to threats and verifiable through testing [15].

That workflow builds on a coauthored minimum-requirements methodology that derives secure-by-design principles from fault-tree analysis of mission-level failure modes, specifically preventing permanent loss of satellite control. The approach identifies single-point-of-failure components across mission segments and maps resilience principles to each, feeding directly into the IEEE P3349 standardization effort [14].

8.1.4 Architecture by Construction

Chapter 5 presents Alcyone, a modular and cyber-resilient flight software system designed to support mission continuity under fault and adversarial conditions. The architecture applies secure-by-design principles at the subsystem level through component-level fault isolation, privilege control, and runtime recovery [16].

Its design derives cyber requirements from threat models and embeds them as enforcement points across interface boundaries. Rust provides memory safety and compile-time ownership checks, while Alcyone’s service decomposition and message interfaces make concurrency, authority, and recovery behavior explicit. SWIL testing and simulated mission workloads exer-

cise fault response logic, monitor requirement compliance, and support end-to-end resilience evaluation. The result is case-study evidence that resilience and verifiability can be developed together when threat models, enforcement boundaries, and evaluation criteria are aligned [16].

8.1.5 Evaluation and Measurement

Chapter 6 defines the evaluation approach: a paired architectural benchmark, repeatable adversarial campaigns, and a metric set—MTTD, MTTR, blast radius, recovery completeness, detection coverage, and a functionality-based resilience ratio R adapted from Weisman et al. [109]—that operationalizes the NIST SP 800-160 Vol. 2 cyber resilience goals (Section 7.1.4). These measures compare design choices by observable resilience properties rather than by checklist compliance.

The quantitative results are bounded but consistent. Across paired adversarial scenarios, Alcyone produced a qualifying onboard detection in 12 of 13 scenarios versus 3 of 13 for the cFS baseline. It also reduced mean MTTD from 7.80 s to 0.83 s, raised mean resilience ratio from 0.722 to 0.941, and reduced mean blast radius from 3.0 mission functions to 1.3. The multi-level functionality study shows where that difference appears: local component disturbance does not necessarily become mission-level loss when service, bus, supervisor, and mission-control boundaries absorb the effect. The sensitivity studies bound the claim by identifying detection floors, phase-dependent consequence, and IDS configuration trade-offs.

The cislunar analysis extends the argument as a boundary condition rather than as a second architecture case study. Longer latency, indirect networking, and distributed operations make the same flight-software questions sharper: which interfaces carry authority, which components can fail or be compromised without cascading, and what evidence remains available when operators cannot intervene immediately.

Across the dissertation, traceability is the durable contribution. Threat analysis informs requirements, requirements define architectural enforcement points, and evaluation tests whether those enforcement points preserve mission function under attack. Credible cyber-resilience

claims about flight software are therefore threat-scoped, requirement-backed, architecture-specific, and evidence-oriented.

8.2 Final Implications

Cyber resilience belongs inside ordinary flight-software engineering practice. Three shifts make that concrete: treating resilience as a design driver alongside safety and reliability, deriving requirements from explicit adversary and mission assumptions, and building architectures that enforce trust boundaries by construction so compromise can be contained and recovery remains meaningful.

This is not only a technical claim. It also requires alignment between mission engineering, acquisition, and standards so security requirements become enforceable, testable obligations rather than aspirational statements. These methods, from attack-surface analysis through threat-derived requirements to architectural enforcement and quantitative evaluation, provide a concrete basis for that alignment [16].

The strongest practical implication is straightforward: flight software is not reliable control software that later receives cybersecurity review. It is a cyber-physical control layer whose failure modes, trust boundaries, and recovery behaviors need to be designed with adversaries in mind from the outset.

References

[1] G. Falco, "Cybersecurity Principles for Space Systems," *Journal of Aerospace Information Systems*, vol. 16, no. 2, pp. 61–70, 2019, publisher: American Institute of Aeronautics and Astronautics _eprint: https://doi.org/10.2514/1.I010693. [Online]. Available: https://doi.org/10.2514/1.I010693

[2] G. Falco, "The Vacuum of Space Cyber Security," in *2018 AIAA SPACE and Astronautics Forum and Exposition*. Orlando, FL: American Institute of Aeronautics and Astronautics, September 2018. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2018-5275

[3] B. Bailey, "Cybersecurity protections for spacecraft: A threat based approach," The Aerospace Corporation, Tech. Rep., 2021. [Online]. Available: https://aerospace.org/sites/default/files/2022-07/DistroA-TOR-2021-01333-Cybersecurity%20Protections%20for%20Spacecraft--A%20Threat%20Based%20Approach.pdf

[4] J. Pavur and I. Moser, "Building a launchpad for impactful satellite cyber-security research," in *Journal of Cybersecurity*, vol. 8, no. 1. Oxford University Press, 2022. [Online]. Available: https://doi.org/10.1093/cybsec/tyac008

[5] U.S.-China Economic and Security Review Commission, "2011 report to congress," U.S.-China Economic and Security Review Commission, Tech. Rep., 2011. [Online]. Available: https://www.uscc.gov/annual-report/2011-annual-report-congress

[6] NASA Office of Inspector General, "Cybersecurity management and oversight at the Jet Propulsion Laboratory," NASA, Tech. Rep. IG-19-022, 2019. [Online]. Available: https://oig.nasa.gov/wp-content/uploads/2024/03/IG-19-022.pdf

[7] N. Boschetti, N. G. Gordon, and G. Falco, "Space cybersecurity lessons learned from the ViaSat cyberattack," in *ASCEND 2022*, ser. ASCEND. American Institute of Aeronautics

and Astronautics, 2022. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2022-4380

[8] R. Ross, V. Pillitteri, R. Graubart, D. Bodeau, and R. McQuaid, “Developing Cyber-Resilient Systems: A Systems Security Engineering Approach,” National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. NIST SP 800-160v2r1, December 2021. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-160v2r1.pdf

[9] J. Curbo and G. Falco, “A research agenda for space flight software security,” in *2023 IEEE 9th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, 2023, pp. 68–77, ISSN: 2836-4171. [Online]. Available: https://ieeexplore.ieee.org/document/10207527

[10] US National Security Agency, “NSA Releases Guidance on How to Protect Against Software Memory Safety Issues,” US National Security Agency, 2022. [Online]. Available: https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidance-on-how-to-protect-against-software-memory-safety-issues/

[11] US Office of the National Cyber Director, *Back to the Building Blocks: A Path Toward Secure and Measurable Software*, The White House, Feb. 2024. [Online]. Available: https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf

[12] J. Curbo and G. Falco, “Cyber resilience in cislunar space: Security strategies for large-scale space infrastructure,” in *2025 IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT)*. IEEE, 2025. [Online]. Available: https://doi.org/10.13140/RG.2.2.15002.56007

[13] J. Curbo and G. Falco, “Attack surface analysis for spacecraft flight software,” in *2024 IEEE 10th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, 2024, pp. 22–30. [Online]. Available: https://doi.org/10.1109/SMC-IT61443.2024.00010

[14] G. Falco, N. Boschetti, A. Viswanathan, B. Bailey, C. Maple, G. Karabulut Kurt, J. Willbold, J. Slay, E. Birrane, D. Logsdon, S. Bennett, W. Ferguson, J. Curbo, J. Sigholm, C. Mehlman, R. Thummala, M. Calabrese, A. Le, K. Tan, and O. Ben Yahia, "Minimum requirements for space system cybersecurity -ensuring cyber access to space," in *2024 IEEE 10th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*. IEEE, 2024. [Online]. Available: https://doi.org/10.1109/SMC-IT61443.2024.00016

[15] J. Curbo and G. Falco, "Testable cyber requirements for space flight software," in *2025 IEEE Aerospace Conference*. IEEE, 2025, pp. 1–20. [Online]. Available: https://ieeexplore.ieee.org/document/11068629

[16] J. Curbo and G. Falco, "Alcyone: A blueprint for secure rust flight software," in *2025 IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT)*. IEEE, 2025. [Online]. Available: https://doi.org/10.13140/RG.2.2.17022.91202

[17] G. J. Holzmann, "Landing a Spacecraft on Mars," *IEEE Software*, vol. 30, no. 2, pp. 83–86, March 2013. [Online]. Available: https://doi.org/10.1109/MS.2013.32

[18] NIST Computer Security Resource Center. (n.d.) attack surface - glossary | CSRC. [Online]. Available: https://csrc.nist.gov/glossary/term/attack_surface

[19] W. A. Wheeler, N. Cohen, J. Betser, and R. M. Ewart, "Cyber Resilient Flight Software for Spacecraft," in *2018 AIAA SPACE and Astronautics Forum and Exposition*. Orlando, FL: American Institute of Aeronautics and Astronautics, September 2018. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2018-5220

[20] NASA, "NASA Software Engineering Requirements," n.d. [Online]. Available: https://nodis3.gsfc.nasa.gov/displayDir.cfm?Internal_ID=N_PR_7150_002D_&page_name=AppendixD

[21] ECSS, *ECSS-E-ST-40C – Software (6 March 2009) | European Cooperation*

for Space Standardization, 2009. [Online]. Available: https://ecss.nl/standard/ecss-e-st-40c-software-general-requirements/

[22] Jet Propulsion Laboratory, "JPL Institutional Coding Standard," n.d. [Online]. Available: https://yurichev.com/mirrors/C/JPL_Coding_Standard_C.pdf

[23] N. Tsamis, B. Bailey, and G. Falco, "Translating Space Cybersecurity Policy into Actionable Guidance for Space Vehicles," in *ASCEND 2021*. American Institute of Aeronautics and Astronautics, November 2021, _eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2021-4051. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.2021-4051

[24] B. Bailey, "Establishing space cybersecurity policy, standards, and risk management practices," The Aerospace Corporation, Tech. Rep., October 2020. [Online]. Available: https://aerospace.org/sites/default/files/2020-10/Bailey%20SPD5_20201010%20V2_formatted.pdf

[25] R. Thummala, E. Rice, and G. Falco, "Why is space cybersecurity unique?" in *Workshop on Security of Space and Satellite Systems (SpaceSec) 2026*. San Diego, CA, USA: Internet Society, February 2026. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/spacesec26-55.pdf

[26] The White House. (2020) Memorandum on space policy directive-5—cybersecurity principles for space systems. [Online]. Available: https://trumpwhitehouse.archives.gov/presidential-actions/memorandum-space-policy-directive-5-cybersecurity-principles-space-systems/

[27] The White House. (2021) Executive order 14028: Improving the nation's cybersecurity. [Online]. Available: https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity

[28] The White House. (2025) Executive order 14144: Strengthening and promoting innovation in the nation's cybersecurity. [On-

line]. Available: https://www.federalregister.gov/documents/2025/01/17/2025-01470/strengthening-and-promoting-innovation-in-the-nations-cybersecurity

[29] Space Systems Critical Infrastructure Working Group, "Recommendations to space system operators for improving cybersecurity," Cybersecurity and Infrastructure Security Agency (CISA), Tech. Rep., Jun. 2024. [Online]. Available: https://www.cisa.gov/resources-tools/resources/recommendations-space-system-operators-improving-cybersecurity

[30] European Union Agency for Cybersecurity (ENISA), "Space threat landscape 2025," Publications Office of the European Union, Tech. Rep., Mar. 2025. [Online]. Available: https://www.enisa.europa.eu/publications/enisa-space-threat-landscape-2025

[31] Bundesamt für Sicherheit in der Informationstechnik. TR-03184 information security for space systems. [Online]. Available: https://www.bsi.bund.de/EN/Themen/Unternehmen-und-Organisationen/Informationen-und-Empfehlungen/IT-Sicherheit-in-Luft-und-Raumfahrt/it-sicherheit-in-luft-und-raumfahrt.html

[32] Ministry of Economy, Trade and Industry (Japan). (2023) Cybersecurity guidelines for commercial space systems, version 1.1. [Online]. Available: https://www.meti.go.jp/shingikai/mono_info_service/sangyo_cyber/wg_seido/wg_uchu_sangyo/pdf/20230331_1e.pdf

[33] Space Information Sharing and Analysis Center. (2025) Space ISAC announces UK global hub during 40th annual space symposium. [Online]. Available: https://spaceisac.org/space-isac-announces-uk-global-hub-during-40th-annual-space-symposium/

[34] Space Information Sharing and Analysis Center. (2025) Global space collaboration strengthened with MOU between space ISAC and french space command. [Online]. Available: https://spaceisac.org/

[35] M. Scholl and T. Suloway, "Introduction to cybersecurity for commercial satellite operations," National Institute of Standards and Technology, Tech. Rep. NIST IR 8270,

2023. [Online]. Available: https://csrc.nist.gov/pubs/ir/8270/final

[36] J. McCarthy, D. Mamula, J. Brule, and K. Meldorf, "Satellite ground segment: Applying the cybersecurity framework to satellite command and control," National Institute of Standards and Technology, Tech. Rep. NIST IR 8401, 2022. [Online]. Available: https://csrc.nist.gov/pubs/ir/8401/final

[37] NIST, *IR 8441: Cybersecurity Framework Profile for Hybrid Satellite Networks (HSN)*, National Institute of Standards and Technology (NIST), 2023. [Online]. Available: https://csrc.nist.gov/pubs/ir/8441/final

[38] NASA, *Space Security:Best Practices Guide (BPG)*, National Aeronautics and Space Administration (NASA), 2024, rev. B. [Online]. Available: https://swehb.nasa.gov/display/SWEHBVD/7.22+-+Space+Security%3A+Best+Practices+Guide

[39] Committee on National Security Systems, "Cybersecurity policy for space systems used to support national security missions," Committee on National Security Systems, Tech. Rep. CNSSP 12, 2025.

[40] Committee on National Security Systems, "Space platform overlay," Committee on National Security Systems, Tech. Rep. CNSSI 1253F Attachment, 2022.

[41] The Aerospace Corporation, "Space segment cybersecurity profile for national security space," The Aerospace Corporation, Tech. Rep. TOR-2023-02161-RevA, 2023. [Online]. Available: https://sparta.aerospace.org/resources/TOR-2023-02161-RevA%20Space%20Segment%20Cybersecurity%20Profile.pdf

[42] IEEE Standards Association. (2024) P3349 - space system cybersecurity working group - home. [Online]. Available: https://sagroups.ieee.org/3349/

[43] Thales Group, "Thales seizes control of ESA demonstration satellite in first cybersecurity exercise of its kind," Press release, Apr. 2023, demonstrated at CYSAT 2023, Paris. [Online]. Available: https://www.thalesgroup.com/en/worldwide/security/press_release/

thales-seizes-control-esa-demonstration-satellite-first

[44] B. Bailey and B. Roeher, “Hacking an On-Orbit Satellite: An Analysis of the CYSAT 2023 Demo,” May 2023. [Online]. Available: https://medium.com/the-aerospace-corporation/hacking-an-on-orbit-satellite-an-analysis-of-the-cysat-2023-demo-ae241e5b8ee5

[45] W. M. Zhang, A. Dai, K. Ryan, D. Levin, N. Heninger, and A. Schulman, “Don’t look up: There are sensitive internal links in the clear on GEO satellites,” in *Proceedings of the 32nd ACM SIGSAC Conference on Computer and Communications Security (CCS ’25)*, Taipei, Taiwan, 2025, distinguished Paper Award. [Online]. Available: https://doi.org/10.1145/3719027.3765198

[46] B. Bailey, “NASA IV&V’s Cyber Range for Space Systems,” February 2019, nTRS Author Affiliations: Tmc Technologies NTRS Meeting Information: Annual Ground System Architectures Workshop (GSAW); 2019-02-25 to 2019-02-28; undefined NTRS Report/Patent Number: GSFC-E-DAA-TN65725 NTRS Document ID: 20190001085 NTRS Research Center: Goddard Space Flight Center (GSFC). [Online]. Available: https://ntrs.nasa.gov/citations/20190001085

[47] Aerospace Corporation, “Space Attack Research & Tactic Analysis (SPARTA),” n.d. [Online]. Available: https://sparta.aerospace.org/

[48] MITRE ATT&CK. (n.d.) Techniques - ICS | MITRE ATT&CK®. [Online]. Available: https://attack.mitre.org/techniques/ics/

[49] D. Adler, D. Miller, and J. Jurgensen, “High adversary tier threat response interdicting cyberspace kill-chain (HAT TRICK): A cybersecurity requirements reference guide for national security systems version 1,” 2023.

[50] M. Segovia, J. Rubio-Hernán, A. R. Cavalli, and J. Garcia-Alfaro, “A survey on cyber-resilience approaches for cyber-physical systems,” *ACM Computing Surveys*, vol. 56, no. 8, pp. 202:1–202:37, 2024. [Online]. Available: https://doi.org/10.1145/3652953

[51] L. Seidel and J. Beier, "Bringing rust to safety-critical systems in space," 2024. [Online]. Available: https://arxiv.org/abs/2405.18135

[52] R. Ross, M. Winstead, and M. McEvilley, "Engineering trustworthy secure systems," National Institute of Standards and Technology, Tech. Rep. NIST Special Publication (SP) 800-160 Vol. 1 Rev. 1, 2022. [Online]. Available: https://csrc.nist.gov/pubs/sp/800/160/v1/r1/final

[53] E. Ear, "Towards principled analysis and mitigation of space cyber risks," Ph.D. dissertation, University of Colorado Colorado Springs, 2025, arXiv:2508.16991. [Online]. Available: https://arxiv.org/abs/2508.16991

[54] M. Starcik, A. Olchawa, R. Fradique, and A. Boulaich, "NASA cFS version aquila software vulnerability assessment," VisionSpace Technologies, Mar. 2025. [Online]. Available: https://visionspace.com/nasa-cfs-version-aquila-software-vulnerability-assessment/

[55] G. Klein, J. Andronick, M. Fernandez, I. Kuz, T. Murray, and G. Heiser, "Formally verified software in the real world," *Communications of the ACM*, vol. 61, no. 10, pp. 68–77, September 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3230627

[56] 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*. Internet Society, 2024. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/spacesec2024-57-paper.pdf

[57] C. Kern, "Safe coding," *Communications of the ACM*, vol. 69, no. 3, pp. 44–55, 2026. [Online]. Available: https://doi.org/10.1145/3795888

[58] C. Kern, "Developer ecosystems for software safety," *Communications of the ACM*, vol. 67, no. 6, pp. 52–60, 2024. [Online]. Available: https://doi.org/10.1145/3651621

[59] M. Souppaya, K. Scarfone, and D. Dodson, "Secure software development framework (SSDF) version 1.1: Recommendations for mitigating the risk of software vulnerabilities,"

National Institute of Standards and Technology, Tech. Rep. NIST Special Publication (SP) 800-218, 2022. [Online]. Available: https://csrc.nist.gov/pubs/sp/800/218/final

[60] Chromium Projects. (n.d.) Memory safety. [Online]. Available: https://www.chromium.org/Home/chromium-security/memory-safety/

[61] J. Vander Stoep and A. Rebert. (2024) Eliminating memory safety vulnerabilities at the source. [Online]. Available: https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html

[62] A. Rebert and C. Kern, "Secure by design: Google's perspective on memory safety," Google Security Engineering, Tech. Rep., 2024. [Online]. Available: https://security.googleblog.com/2024/03/secure-by-design-googles-perspective-on.html

[63] V. Yodaiken, "How ISO C became unusable for operating systems development," in *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*, October 2021, pp. 84–90, arXiv:2201.07845 [cs]. [Online]. Available: http://arxiv.org/abs/2201.07845

[64] US Cybersecurity and Infrastructure Security Agency, "Secure by Design | CISA," US Cybersecurity and Infrastructure Security Agency, 2023. [Online]. Available: https://www.cisa.gov/securebydesign

[65] W. Snavely, C. Meyers, B. Runyon, C. Inacio, M. Riley, and J. D. Lareau, "Flight Software Programming Language Selection: A Security Perspective," in *2018 AIAA SPACE and Astronautics Forum and Exposition*. Orlando, FL: American Institute of Aeronautics and Astronautics, September 2018. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2018-5397

[66] Ferrous Systems, "Ferrocene: Safety-qualified rust toolchain," 2024, qualified for ISO 26262 (ASIL D) and IEC 61508 (SIL 4). [Online]. Available: https://ferrocene.dev/

[67] R. Mueller. (2024) sat-rs | IRS satellite division software projects. [Online]. Available:

https://absatsw.irs.uni-stuttgart.de/projects/sat-rs/

[68] R. Ross. (2023) Ron ross quote on cyber resilience. [Online]. Available: https://www.linkedin.com/feed/update/urn:li:activity:7147993287511982080/

[69] L. Kohnfelder and P. Garg, "The threats to our products," *Microsoft Interface*, 1999. [Online]. Available: https://shostack.org/files/microsoft/The-Threats-To-Our-Products.docx

[70] M. Morana and T. Ucedavelez. (2011) Risk analysis of banking malware attacks. [Online]. Available: https://www.slideshare.net/marco_morana/owasp-app-seceu2011version1

[71] D. Papp, Z. Ma, and L. Buttyan, "Embedded systems security: Threats, vulnerabilities, and attack taxonomy," in *2015 13th Annual Conference on Privacy, Security and Trust (PST)*, 2015, pp. 145–152. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7232966

[72] A. Easwaran, A. Chattopadhyay, and S. Bhasin, "A systematic security analysis of real-time cyber-physical systems," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017, pp. 206–213. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7858321

[73] F. Sagstetter, M. Lukasiewycz, S. Steinhorst, M. Wolf, A. Bouard, W. R. Harris, S. Jha, T. Peyrin, A. Poschmann, and S. Chakraborty, "Security challenges in automotive hardware/software architecture design," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2013, pp. 458–463, ISSN: 1530-1591. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6513548

[74] J. Edwards, A. Kashani, and G. Iyer, "Evaluation of software vulnerabilities in vehicle electronic control units," in *2017 IEEE Cybersecurity Development (SecDev)*, 2017, pp. 83–84. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8077811

[75] K. Zhang and A. Olmsted, "Examining Autonomous Vehicle Operating Systems Vulnerabilities using a Cyber-Physical Approach," in *2021 IEEE International Intelligent*

Transportation Systems Conference (ITSC), 2021, pp. 976–981. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9564848

[76] T. Kiesling, M. Krempel, J. Niederl, and J. Ziegler, “A model-based approach for aviation cyber security risk assessment,” in *2016 11th International Conference on Availability, Reliability and Security (ARES)*, 2016, pp. 517–525. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7784614

[77] E. Habler, R. Bitton, and A. Shabtai, “Assessing aircraft security: A comprehensive survey and methodology for evaluation,” *ACM Computing Surveys*, vol. 56, no. 4, pp. 96:1–96:40, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3610772

[78] CISA. (n.d.) Seven steps to effectively defend industrial control systems_s508c.pdf. [Online]. Available: https://www.cisa.gov/sites/default/files/documents/Seven%20Steps%20to%20Effectively%20Defend%20Industrial%20Control%20Systems_S508C.pdf

[79] G. O. Passmore and D. Ignatovich, “Formal verification of financial algorithms,” in *Automated Deduction – CADE 26*, ser. Lecture Notes in Computer Science, L. de Moura, Ed. Springer International Publishing, 2017, pp. 26–41. [Online]. Available: https://doi.org/10.1007/978-3-319-63046-5_3

[80] The RTEMS Project. (2023) 1. introduction — RTEMS user manual 5.3 (10th february 2023) documentation. [Online]. Available: https://docs.rtems.org/releases/rtems-5.3/user/overview/index.html#real-time-executive

[81] E. Kovacs. (2018) FreeRTOS vulnerabilities expose many systems to attacks. [Online]. Available: https://www.securityweek.com/freertos-vulnerabilities-expose-many-systems-attacks/

[82] The RTEMS Project. (2023) 8.7. dynamic loader — RTEMS user manual 5.3 (10th february 2023) documentation. [Online]. Available: https://docs.rtems.org/releases/rtems-5.3/user/exe/loader.html

[83] RTEMS Documentation Project. (2023) RTEMS shell guide (5.3). — RTEMS shell guide 5.3 (10th february 2023) documentation. [Online]. Available: https://docs.rtems.org/releases/rtems-5.3/shell/index.html

[84] CrowdStrike. (n.d.) What are living off the land (LOTL) attacks? [Online]. Available: https://www.crowdstrike.com/cybersecurity-101/living-off-the-land-attacks-lotl/

[85] A. Viswanathan, B. Bailey, K. Tan, and G. Falco, "Secure-by-component: A system-of-systems design paradigm for securing space missions," 2024. [Online]. Available: https://indico.esa.int/event/528/attachments/5988/10194/Secure_by_component_A_System_of_systems_Design_Paradigm_for_Securing_Space_Missions.pdf

[86] J. T. Force, "Security and privacy controls for information systems and organizations," National Institute of Standards and Technology, Tech. Rep. NIST Special Publication (SP) 800-53 Rev. 5, 2020. [Online]. Available: https://csrc.nist.gov/pubs/sp/800/53/r5/upd1/final

[87] CCSDS, *350.0-G: The Application of Security to CCSDS Protocols*, The Consultative Committee for Space Data Systems (CCSDS), 2019, issue 3. [Online]. Available: https://public.ccsds.org/Pubs/350x0g3.pdf

[88] CCSDS, *355.0-B: Space Data Link Security Protocol*, The Consultative Committee for Space Data Systems (CCSDS), 2022, issue 2. [Online]. Available: https://public.ccsds.org/Pubs/355x0b2.pdf

[89] NASA Goddard Space Flight Center. Core flight system. [Online]. Available: https://cfs.gsfc.nasa.gov/

[90] NASA Jet Propulsion Laboratory. (n.d.) Meet the open-source software powering NASA's ingenuity mars helicopter. [Online]. Available: https://www.jpl.nasa.gov/news/meet-the-open-source-software-powering-nasas-ingenuity-mars-helicopter

[91] C. S. Păsăreanu and W. Visser, "Symbolic Execution and Model Checking for Testing,"

in *Hardware and Software: Verification and Testing*, ser. Lecture Notes in Computer Science, K. Yorav, Ed. Berlin, Heidelberg: Springer, 2008, pp. 17–18. [Online]. Available: https://doi.org/10.1007/978-3-540-77966-7_5

[92] A. Groce, K. Havelund, G. Holzmann, R. Joshi, and R.-G. Xu, "Establishing flight software reliability: testing, model checking, constraint-solving, monitoring and learning," *Annals of Mathematics and Artificial Intelligence*, vol. 70, no. 4, pp. 315–349, April 2014. [Online]. Available: https://doi.org/10.1007/s10472-014-9408-8

[93] J. Willbold, M. Schloegel, F. Göhler, T. Scharnowski, N. Bars, S. Wörner, N. Schiller, and T. Holz, "Scaling software security analysis to satellites: Automated fuzz testing and its unique challenges," in *2024 IEEE Aerospace Conference*, 2024, pp. 1–12, ISSN: 1095-323X. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10521316

[94] K. Fisher, J. Launchbury, and R. Richards, "The HACMS program: using formal methods to eliminate exploitable bugs," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 375, no. 2104, p. 20150401, September 2017, publisher: Royal Society. [Online]. Available: https://royalsocietypublishing.org/doi/10.1098/rsta.2015.0401

[95] A. Sharma, S. Sharma, S. R. Tanksalkar, S. Torres-Arias, and A. Machiry, "Rust for embedded systems: Current state and open problems," in *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*, Salt Lake City, UT, USA, 2024. [Online]. Available: https://doi.org/10.1145/3658644.3690275

[96] CCSDS, *133.0-B-2: Space Packet Protocol*, The Consultative Committee for Space Data Systems (CCSDS), 2020, issue 2, Errata 2. [Online]. Available: https://ccsds.org/Pubs/133x0b2e2.pdf

[97] CCSDS, *232.0-B-4: TC Space Data Link Protocol*, The Consultative Committee for Space Data Systems (CCSDS), 2021, issue 4, Corrigendum 1. [Online]. Available:

https://ccsds.org/Pubs/232x0b4e1c1.pdf

[98] CCSDS, *132.0-B-3: TM Space Data Link Protocol*, The Consultative Committee for Space Data Systems (CCSDS), 2021, issue 3. [Online]. Available: https://ccsds.org/Pubs/132x0b3.pdf

[99] CCSDS, *727.0-B-5: CCSDS File Delivery Protocol (CFDP)*, The Consultative Committee for Space Data Systems (CCSDS), 2020, issue 5. [Online]. Available: https://stage.tksc.jaxa.jp/ccsds/docs/files/bluebook/sis/727_0_b_5.pdf

[100] CCSDS, *876.0-B-1: XML Specification for Electronic Data Sheets for Onboard Devices and Components*, The Consultative Committee for Space Data Systems (CCSDS), 2019, issue 1. [Online]. Available: https://ccsds.org/Pubs/876x0b1.pdf

[101] CCSDS, *660.0-B-2: XML Telemetric and Command Exchange (XTCE)*, The Consultative Committee for Space Data Systems (CCSDS), 2020, issue 2. [Online]. Available: https://ccsds.org/Pubs/660x0b2.pdf

[102] ECSS, *ECSS-E-ST-70-41C: Space Engineering—Telemetry and Telecommand Packet Utilization*, European Cooperation for Space Standardization (ECSS), 2016. [Online]. Available: https://ecss.nl/wp-content/uploads/2016/06/ECSS-E-ST-70-41C15April2016.pdf

[103] CCSDS, *732.1-B-2: Unified Space Data Link Protocol*, The Consultative Committee for Space Data Systems (CCSDS), 2021, issue 2. [Online]. Available: https://stage.tksc.jaxa.jp/ccsds/docs/files/bluebook/sls/732_1_b_2.pdf

[104] A. Lyons, K. McLeod, H. Almatary, and G. Heiser, "Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time," in *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, 2018, pp. 1–13.

[105] Google Security Blog. (2025) Android's journey to memory safety. [Online]. Available: https://security.googleblog.com/2025/11/androids-journey-to-memory-safety.html

[106] Ferrous Systems, “Ferrocene language specification,” https://spec.ferrocene.dev/, 2024.

[107] R. Bulusu, “SoK: Resilience and fault tolerance in cyber-physical systems,” arXiv:2512.20873, 2025. [Online]. Available: https://arxiv.org/abs/2512.20873

[108] A. Kott, M. J. Weisman, and J. Vandekerckhove, “Mathematical modeling of cyber resilience,” in *MILCOM 2022 - 2022 IEEE Military Communications Conference (MILCOM)*, 2022, pp. 849–854, ISSN: 2155-7586. [Online]. Available: https://ieeexplore.ieee.org/document/10017731

[109] M. J. Weisman, A. Kott, J. E. Ellis, B. J. Murphy, T. W. Parker, S. Smith, and J. Vandekerckhove, “Quantitative measurement of cyber resilience: Modeling and experimentation,” *ACM Transactions on Cyber-Physical Systems*, vol. 9, no. 1, pp. 1–27, 2025. [Online]. Available: https://doi.org/10.1145/3703159

[110] MITRE Corporation, “Cyber resiliency engineering framework (CREF) navigator,” MITRE, Tech. Rep., 2022, includes the Cyber Resilience Assessment and Management (CRAM) methodology. [Online]. Available: https://cref.mitre.org/

[111] S. Tse, J. Lucas, J. Marshall, and L. Nguyen, “NASA operational simulator for small satellites (NOS3): Advancements in spacecraft simulation,” in *Proceedings of the Small Satellite Conference*, Logan, UT, 2018.

[112] The Aerospace Corporation, “SPARTEND: Integrating space cyber threat knowledge with autonomous detection,” Aerospace Corporation Technical Article, 2024, presented at DEF CON 32 Aerospace Village. [Online]. Available: https://aerospace.org/article/aerospaces-spartend-integrates-space-cyber-threat-knowledge-autonomous-detection

[113] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, “Fault injection techniques and tools,” *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[114] J. Carreira, H. Madeira, and J. G. Silva, “Xception: A technique for the experimental evaluation of dependability in modern computers,” *IEEE Transactions on Software Engineering*,

vol. 24, no. 2, pp. 125–136, 1998.

Appendix A

Acronyms

This appendix lists the acronyms used throughout the dissertation. Where an acronym is introduced in the text, the long form appears on first use and the short form is used thereafter.

Acronym	Expansion
ACL	access control list
API	application programming interface
BSI	German Federal Office for Information Security
BSP	board support package
CCSDS	Consultative Committee for Space Data Systems
CFDP	CCSDS File Delivery Protocol
cFE	core Flight Executive
cFS	core Flight System
CISA	Cybersecurity and Infrastructure Security Agency
CPS	cyber-physical systems
C&DH	command and data handling
ECSS	European Cooperation for Space Standardization
FDIR	fault detection, isolation, and recovery
FPR	false positive rate
FSW	flight software

Acronym list, continued.

Acronym	**Expansion**
HAT TRICK	High Adversary Tier Threat Response Interdicting Cyberspace Kill-chain
ICS	industrial control systems
IDS	intrusion detection service
IEEE	Institute of Electrical and Electronics Engineers
IPC	inter-process communication
LEO	low Earth orbit
MTTD	mean time to detect
MTTR	mean time to recover
NASA	National Aeronautics and Space Administration
NIST	National Institute of Standards and Technology
NOS3	NASA Operational Simulator for Small Satellites
OSAL	Operating System Abstraction Layer
POSIX	Portable Operating System Interface
RC	recovery completeness
ROSAT	Röntgensatellit
RTEMS	Real-Time Executive for Multiprocessor Systems
RTOS	real-time operating system
SCADA	supervisory control and data acquisition
SDLS	Space Data Link Security

Acronym list, continued.

Acronym	**Expansion**
seL4	secure L4 microkernel
SPARTA	Space Attack Research and Tactic Analysis
STRIDE	Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege
SWIFI	software-implemented fault injection
SWIL	software-in-the-loop

Appendix B

Top-Level Cyber Resilience Requirements

This appendix lists the 27 top-level functional requirements (FR) that govern the Alcyone implementation evaluated in Chapters 5–6. The set was generated from the sextant artifact-management tool described in Section 4.4.6 and represents one view of the linked artifact graph that includes 89 individual shall-statements in total: 27 top-level functional requirements listed here, plus 7 ground-segment interface requirements not reproduced in this appendix, plus 55 sub-requirements that decompose the top-level entries into concrete enforcement obligations.

The C&DH walk in Section 4.3 of Chapter 4 illustrates the secure-by-component methodology in its originally published form, organized by C&DH subcomponent (Command Reception, Telemetry Generation, Data Storage, Health and Status). As Alcyone was implemented, the requirement set was reshaped from C&DH subcomponents to architectural roles — top-level functional requirements that name the property each service must enforce — and the resulting set is what this appendix records.

Cross-references to architectural views, services, fault definitions, threat actors, and verification evidence are captured in the sextant artifact graph and not reproduced here. Each top-level requirement also has sub-requirements that elaborate concrete enforcement obligations.

- **FR-01 — Fault Detection and Recovery.** The system shall detect service faults and execute bounded recovery actions.

- **FR-02 — Service Isolation and Containment.** The system shall isolate services such that faults or compromises in one service cannot propagate to others.

- **FR-03 — Explicit Authority.** The system shall centralize command, actuation, and recovery authority in designated services.

- **FR-04 — Boundary Validation.** The system shall validate all external inputs at system boundaries and reject malformed or unauthorized data.
- **FR-05 — Observability.** The system shall emit structured events and telemetry for all security-relevant and non-nominal behavior.
- **FR-06 — Bounded Resources.** The system shall bound all queues, buffers, and retry attempts with explicit backpressure policies.
- **FR-07 — Message-Based Communication.** The system shall use message passing as the sole inter-service communication mechanism.
- **FR-08 — Mode Management.** The system shall support explicit operational modes with defined transitions and per-mode policies.
- **FR-09 — Configuration Integrity.** The system shall protect configuration data integrity and validate changes before application.
- **FR-10 — External Interface Protocol.** The system shall use CCSDS Space Packets for all external communication.
- **FR-11 — Performance Observability.** The system shall support measurement and observation of time and space performance characteristics.
- **FR-12 — Internal Message Validation.** The system shall validate internal messages at service boundaries to detect and reject unauthorized or malformed inter-service communication.
- **FR-13 — Stored Data Integrity.** The system shall protect the integrity of persistent data and detect tampering.
- **FR-14 — Command Processing.** The system shall receive, parse, validate, authorize, and dispatch commands from external sources.

- **FR-15 — Telemetry Generation.** The system shall collect, format, and transmit telemetry to external consumers.
- **FR-16 — Event Management.** The system shall aggregate, filter, and forward structured events from all services.
- **FR-17 — Health Monitoring.** The system shall periodically assess and report the health status of all monitored services.
- **FR-18 — Sensor Data Ingress.** The system shall receive, validate, and distribute sensor data from external plant interfaces.
- **FR-19 — Actuator Control.** The system shall receive, validate, and execute actuation commands with sole authority via the unified device manager (DEV).
- **FR-20 — Supervisor Operations.** The system shall manage system modes, aggregate health state, coordinate recovery, and serve as the trust anchor.
- **FR-21 — Unified Detection Pipeline.** The system shall process fault events and threat indicators through a unified detection ingestion path before action decisions are made.
- **FR-22 — Deterministic Fault Detection (FDIR).** The system shall provide a dedicated fault detection and isolation service (FDIR) that implements configurable per-service threshold state machines and issues fault directives to SUP.
- **FR-23 — Behavioral Anomaly Detection (IDS).** The system shall provide a dedicated intrusion detection service (IDS) that performs cross-service behavioral pattern analysis over time windows and issues threat assessments with confidence levels to SUP.
- **FR-24 — SUP Decision Fusion.** The SUP service shall fuse inputs from three channels — raw fault reports from EVT, FDIR fault directives, and IDS threat assessments — and apply a decision policy to determine recovery and prevention actions.

- **FR-25 — SUP Prevention Authority.** The SUP service shall have prevention authority — the ability to proactively limit system exposure before a fault or attack fully materializes — including rate limit adjustment, subscription revocation, and load shedding.
- **FR-26 — Detection Decision Observability.** The system shall make all detection, suppression, and action decisions observable via telemetry — including FDIR threshold transitions, IDS assessments with confidence levels, SUP fusion outcomes, and any suppression or rate-limiting applied by detection services.
- **FR-27 — Detection Service Independence.** The FDIR and IDS detection services shall be failure-independent: a fault or crash in one detection service shall not impair the operation of the other, and SUP shall continue to operate using whichever detection input remains available.

Appendix C

Curriculum Vitae

Education

Graduate Certificate, Applied and Computational Mathematics 2024

Johns Hopkins University Whiting School of Engineering, Baltimore, MD

Master of Science, Computer Science 2013

Capitol Technology University, Laurel, MD

Bachelor of Science, Computer Science 2003

Henderson State University, Arkadelphia, AR

Professional Experience

Group Chief Scientist, Constrained Cyber Solutions Group 2024–Present

Johns Hopkins University Applied Physics Laboratory, Laurel, MD

Group Chief Engineer, Cyber Warfare Systems Group 2021–2024

Johns Hopkins University Applied Physics Laboratory, Laurel, MD

Cybersecurity Engineer 2017–2021

Johns Hopkins University Applied Physics Laboratory, Laurel, MD

Cybersecurity Engineer 2010–2017

MITRE Corporation, Fort Meade, MD

Cyberspace Operations Officer, United States Air Force 2004–2010

Teaching

Guest Lecturer, Space and International Public Policy 2024–Present

Johns Hopkins School of Advanced International Studies

Guest Lecturer, Space Systems Engineering Cybersecurity 2024–Present

Johns Hopkins Engineering for Professionals

Standards and Service

Vice-Chair, Ground Segment Subcommittee 2024–Present

IEEE P3536 Working Group on Space System Cybersecurity Design Standards

Program Committee Member 2025–Present

IEEE Workshop on Security and Resiliency of Critical Infrastructure and Space Technologies (SR-CIST)

Program Committee Member 2026–Present

IEEE International Conference on Resilience and Integrated Security for Space and Critical Systems

Publications

1. J. Curbo and G. Falco, "Cyber Resilience in Cislunar Space: Security Strategies for Large-Scale Space Infrastructure," in *Proc. IEEE 11th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, Los Angeles, CA, Jul. 2025.

2. J. Curbo and G. Falco, "Alcyone: A Blueprint for Secure Rust Flight Software," in *Proc. IEEE 11th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, Los Angeles, CA, Jul. 2025.

3. J. Curbo and G. Falco, "Testable Cyber Requirements for Space Flight Software," in *Proc. 2025 IEEE Aerospace Conference*, Big Sky, MT, Mar. 2025.

4. M. Calabrese, J. Curbo, and G. Falco, "A Software Defined Networking Architecture for Time Triggered Ethernet in Space Systems," in *Proc. 2024 IEEE International Conference on Wireless for Space and Extreme Environments (WiSEE)*, Daytona Beach, FL, Dec. 2024.

5. G. Falco, N. Boschetti, A. Viswanathan, B. Bailey, C. Maple, G. K. Kurt, J. Curbo, *et al.*, "Minimum Requirements for Space System Cybersecurity—Ensuring Cyber Access to Space," in *Proc. IEEE 10th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, pp. 78–88, Mountain View, CA, Jul. 2024.

6. J. Curbo and G. Falco, "Attack Surface Analysis for Spacecraft Flight Software," in *Proc. IEEE 10th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, pp. 22–30, Mountain View, CA, Jul. 2024.

7. J. Curbo and G. Falco, "A Research Agenda for Space Flight Software Security," in *Proc. IEEE 9th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, pp. 68–77, Pasadena, CA, Jul. 2023.