EasyChair Preprint

№ 10604

A Research Agenda for Space Flight Software Security

James Curbo and Gregory Falco

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

July 27, 2023

A Research Agenda for Space Flight Software Security

James Curbo
Whiting School of Engineering
Johns Hopkins University
Baltimore, USA
jcurbo1@jhu.edu

Gregory Falco
Institute for Assured Autonomy
Johns Hopkins University
Baltimore, USA
falco@jhu.edu

***Abstract*—Space flight software is no longer a closely guarded secret for space vehicle developers, owners and operators - it is open-sourced and available as a commercial-off-the-shelf module. Despite its wide availability, limited security research has been conducted on flight software in an unclassified environment. This paper proposes a research agenda that outlines critical challenges for space flight software and proposes a series of research and development efforts that could ultimately aid in developing inherently secure space vehicles.**

***Index Terms*—space flight software, space cybersecurity, flight software security**

I. INTRODUCTION

Flight software is fundamental to the successful performance of any space vehicle's mission. The reliability of flight software is not a new topic and has been studied extensively for the past decades through the lens of quality assurance, fault tolerance and fail-safe operations, with particular attention to instrumenting flight software with layers of redundancy. Despite attention to fault management principles and practice, there has been limited attention to the cybersecurity of flight software. The principal difference between fault tolerance for flight software and security challenges for the same is that fault tolerance assumes faults are probabilistic in nature and that failures will occur in a predicable order from predictable environmental effects. Cybersecurity threats to flight software are propagated by an intelligent adversary that may be actively engaging with the flight software, despite fail-safe mechanisms or available defenses, intentionally stressing its processes in an unanticipated fashion. An attacker's pursuit or next move is not as predictable as an environment-propagated failure.

While the flight software community has historically operated under the guise of security by obscurity, the increasingly open-source and commercial-off-the-shelf (COTS) availability of flight modules have erased any perceived security benefits. NASA's core Flight System (cFS) and NASA's Jet Propulsion Laboratory's F' flight software is readily available for adversaries and security researchers alike to explore, which forces a public discourse about space flight software security practices and requirements for the 'new' space era. This paper presents a research agenda for flight software security, discussing the robust related research conducted thus far in related fields and describing opportunities to improve the security posture and cyber resilience of flight software against future threats.

II. CYBER RESILIENCE AND FLIGHT SOFTWARE

A. *Defining Flight Software*

Space missions can use software in many different places in a typical space vehicle architecture. For this paper, we will focus on the core of a space vehicle's software, usually known as "flight software," made up of the real-time operating system (RTOS) and mission applications running on a space vehicle's primary computer. This is a necessarily generic definition due to the wide variety of computing architectures used in space vehicles. Flight software implements a variety of fundamental space vehicle functionality, such as command and data handling, fail-safe mode operations, guidance, navigation, managing instruments and payloads, controlling subsystems like thermal management and power, and managing communications. Many other types of space vehicle-related software exist, including embedded firmware within hardware components, dedicated software running within instruments and hosted payloads that interface with the space vehicle's primary flight software. They are not the focus of this paper, but deserve scrutiny as well.

B. *Why is Cyber Resilience Necessary?*

Space is an increasingly important landscape for national security, national policy, economics, science, and national prestige; this has led to an increase in the recognition by governments of the need to protect and defend critical space systems, one example being recognition in the US as critical national infrastructure [1]. An important part of this protective landscape is recognition of the need for increased cyber resilience and cybersecurity in space systems, both in hardware and software [2] [3].

Cyber resilience is defined by NIST [4, p. 9] as the ability to anticipate, withstand, recover and adapt to adverse conditions, stresses, attacks, or compromises that use or are enabled by cyber resources; that is, information systems, computing architectures, networking and other interconnected electronic devices . These resources can be attacked by malicious outside entities who seek to damage, degrade, deny, or otherwise impair the ability of the system to perform its intended function. Attacks can take many forms and are dependent on the technology involved and the desired effect; this is also true of mitigations.

Historically, space vehicles have not been a prominent target of cyber attack, so their developers have not built them with cyber resilience in mind. There is lack of cybersecurity standards focused on the unique challenges of the space domain and space-focused systems [5] [6]. Organizations developing flight software do not usually specialize or have significant expertise in cybersecurity or secure software engineering practices. The nature of space mission development yields a wide variety of organizations' involvement in developing flight software, where many of these organizations have in-house software engineering groups dedicated to their flight software needs. Some are civilian space agencies focused on scientific missions and human spaceflight, such as NASA and ESA. These organizations oversee many space efforts with a wide variety of goals. Overlapping with this category are universities and other research establishments designing their own missions. Some of these organizations are developing their missions for civilian space agencies, and others are simply pursuing a line of research. Another area is the military and its associated defense contractors. Like scientific missions, military missions have very specific requirements that vary among mission types, and so many defense contractors develop their flight software in-house and for specific missions or related mission sets. Finally, commercial companies who develop space vehicles must develop flight software tailored to their needs and the needs of their customers. All of these organizations need to use flight software, whether they develop it in house, modify an existing software stack, buy it from someone else, or use open source software. These organizations all have different needs when it comes to cyber resilience, which must be discovered, documented and integrated into the development process.

C. Threats Against Flight Software

An important part of building cyber resilience into systems is understanding the threat landscape. This encompasses the known adversary organizations or types of organizations that may seek to target a system, as well as the tactics and techniques that an adversary can use to reconnoiter, exploit, or maliciously affect a system.

Flight software is a prime target for attack by malicious parties. Because of the central nature of the flight software for operation of the space vehicle, access to it can lead to partial or total control of space vehicle functions, which adversaries can co-opt for their own goals. Like any other piece of software, flight software is imperfect and vulnerabilities will probably exist in currently used software components and languages. Since authorized users access them remotely, a secure ground system does not necessarily mean a space vehicle is secure as well. In addition, the growing complexity of software architectures on space vehicles, and increasing use of common software platforms instead of custom-built software components, means that there are more possible pathways for an adversary to take to perform an attack on the space vehicle's systems and introduces a supply-chain aspect to securing these systems.

There is a lack of open source literature about cyber attacks against space vehicles. The Space Attack Research & Tactic Analysis (SPARTA) project [7] was recently started to document and categorize threats against space vehicles but is in its early stages and needs validation and extension.

D. Moving Towards Cyber Resilience

Organizations developing space missions must consider cybersecurity and adversary activity and design their systems accordingly. Cyber attackers do not limit their activities to military targets. Securing one portion of a system alone is not sufficient; mission designers must incorporate resilience against attack into the design of the complete system, including both software and hardware. NIST has released comprehensive guidance for developing cyber-resilient systems [4], however, developers implement few of these recommendations in flight software, much less the entirety of a space system. A potential reason for the meager adoption of such resilience guidelines is that they are not written specifically for space system developers as is the intention of future international space standards [8].

The NIST guidelines offer a broad set of objectives not specific to space systems. These include the following: prevention or avoidance of adverse conditions; preparation of course of action to address adversity; maximizing the duration and viability of essential functions during adversary; constraining damage; and providing the ability to quickly or reliably restore service after adverse conditions. Bailey lays out a set of cyber resilience principles that can be applied to the development of space vehicles [9]. These are being robust, being opaque, constraining behavior, and being responsive.

Developers can apply all these principles to flight software, but the robustness and constraint principles are the most applicable to the software domain. When applied to flight software design, these principles can provide ways to provide resistance against attacks, containment of attacks as they move through components, or mitigation against successful attacks. Flight software must constrain or eliminate unsafe or undesired behavior such that an attacker can not co-opt or leverage system components to produce harmful effects.

III. Design Considerations for Cyber Resilience

To understand how and where to add cyber resilience to flight software systems, we have taken a comprehensive look at how these systems are currently developed and fielded and examined their current ability to address cyber threats. This analysis resulted in a set of design considerations that we use to frame the problem and understand how to address fundamental cybersecurity challenges in various sections of flight software development and operations. At the same time, these considerations also allowed us to understand what was missing from the development landscape, so that we could produce a research agenda to explore creation of new techniques or application of techniques to areas where they were previously not fielded.

A. *Requirements and Quality Assurance*

Space vehicle development projects, including flight software, typically have associated sets of detailed requirements and engineering guidelines. These are designed to ensure the software is developed to its desired purpose, meets the needs of mission owners, includes required technical functions, and will execute according to its intended design with risk that is mitigated and appropriately traded off. Examples of these types of standards include NASA's Software Engineering Requirements [10], the European Space Agency (ESA) code standards [11], the Jet Propulsion Laboratory (JPL) F' flight software system [12], and JPL's standards for using the C programming language [13].

Software quality is an important and highly developed discipline within software engineering, especially for flight software. Flight software developers build their systems with a focus on reliability and fault tolerance and are often on the cutting edge of various quality related software engineering practices. Holzmann provides a detailed case study of software quality in the Mars Science Laboratory mission and highlights the need for reliable code and detailed software engineering processes [14].

Since many existing quality standards focus on ensuring the software meets its intended design, they do not specifically include understanding, measuring or requiring resilience against cyber attacks or related security concerns. Cybersecurity requirements, when they do exist for flight software systems, are often extremely specific, focused on compliance, and lack traceability to mission needs or specific cyber threats [15]. Because the pace of development of cyber threats is often faster than the development of complex systems, these systems can be insecure immediately upon going into operations. These reasons make existing quality standards insufficient, but not unnecessary, for addressing the security of a flight software system. Developers should apply quality standards to mitigations used to make a flight software system more resilient to cyber attack, and disallow harmful features or design patterns that are known to be leveraged by adversaries.

B. *Security Analysis and Testing*

One way to augment existing quality assurance processes to increase security is to leverage security-focused and adversarial testing and evaluation methods to look for cyber-related weaknesses. These tools and methods take many forms and can be applied at distinct steps of the systems engineering lifecycle. During the development phase, code analysis tools can be used to scan and identify weaknesses in source code and binaries. Today, many of these tools are used for quality assurance by searching for flaws in code that may result in incorrect program behavior. While important, this does not go far enough to evaluate the security of the system.

Static code analysis is straightforward to implement for many code bases and static code analyzers are available for many languages, both commercially and open-source. However, static code analysis often provides false positives and hard to understand results. Wheeler *et al.* present a detailed discussion of the use of static code analysis in flight software [16]. Dynamic analysis tools can bridge some of the gaps in static analysis tools by providing varying test inputs and running code to find problems, but dynamic analysis is more difficult to instantiate and requires dedicated test setups and properly emulated system inputs. One example of a comprehensive dynamic analysis setup is NASA's Jon McBride Software Testing and Research (JSTAR) Laboratory which provides IV&V services to NASA missions [17]. NASA has also developed large scale cyber ranges for IV&V testing and training of cyber defenders [18].

Another set of approaches focuses on emulating the adversary. Penetration testing and red teaming refer to a specific style of testing that seeks to employ known adversary techniques to find and exploit weaknesses in running systems. Developers can use these testing methods at various steps of the development lifecycle, either as feedback while they write code (in the form of assessments used incrementally in the development process), test events during integration activities that attack a component or set of components of a running system, all the way to full-up *in situ* testing of a fully developed and integrated system that is feature-complete. In all cases, these methods rely on accurate and timely threat intelligence of cyber adversary techniques and tools.

Both sets of testing methods presented above, code analysis and penetration testing/red teaming, are empirical techniques based on observation and experimentation to uncover defects that adversaries use to attack systems. While these techniques have proven useful, they require a large body of knowledge to know where to look and what to do. In addition, these empirical techniques do not prove correctness of a specific software implementation, instead they help verify that either specific attack patterns are not possible or are difficult to execute. More performant code analysis tools and efficient red teaming methodologies only decrease the uncertainty around defects, but do not remove them completely.

C. *Security Aspects of Architectural Design*

Designing software architectures with security-specific features in mind is a way to limit unsafe and unwanted system behavior at the lowest levels of the overall software architecture and earliest stages of development. One method is to use a design language to specify system behaviors and interfaces and verify that software components follow the specification. This is the approach of the Architecture Analysis and Design Language (AADL), which is used in the avionics world and is beginning to be applied to space vehicle design [19] [20].

Another approach is to enforce separation of concerns and isolate system components so that they can only interact with each other when necessary and enforcing that by secure software design such as formal verification. One example of this approach is the seL4 microkernel [21]. Mission developers can use this fully verified, provably secure low-level operating system as a foundation where the various components can only communicate as allowed. This allows the usage of possibly untrusted software components, or components leveraging unsafe technologies, since they will not be able to breach the walls of their containment due to the secure nature of the seL4 isolation environment. Another example of this approach, from ESA, presents a methodology for developing functional requirements implementing security isolation and partitioning (in time and space) for flight software [22].

A prominent example of secure architectures and use of formal methods for the development of secure software is the DARPA High-Assurance Cyber Military Systems program. This program developed software for two aircraft platforms (a quadcopter and a full helicopter) using verified operating systems providing isolation and application code developed using memory-safe programming techniques. Fisher describes the ways formal methods were used in application development during the project [23], and Klein *et al.* detail the use of the seL4 microkernel and how it was used to derive security guarantees for the entire architecture [24].

D. Security of the RTOS

Real time operating systems are foundational to flight software. Many flight software projects choose to use an out-of-the-box, commonly used RTOS, with popular options being vxWorks, FreeRTOS, and RTEMS. Because of the real-time processing needs of these types of embedded systems, and these RTOSes are typically written in low-level languages such as C or C++. However, these languages have unsafe characteristics if not used correctly. Methods exist to check the RTOS code base to ensure that common problems are avoided, such as static and dynamic code analysis methods, model checking, and implementation of critical portions of the RTOS in formally verified languages. Many RTOS projects have implemented several of these solutions, but they are typically targeted at quality-related issues, not cyber resilience issues, due to current practice as mentioned elsewhere. Because of the core nature of the RTOS, it is a key target for adversaries looking for weaknesses, and any vulnerability that is discovered may cause an outsized impact. Therefore, flight software developers should keep security and adversary activity in mind and highly scruitinze the RTOS, hardware interfaces (such as platform support kits) and the interface to the rest of the flight software stack.

E. Security of Mission Applications and Hosted Payloads/Instruments

Mission applications running on top of the RTOS must also be built securely and have unique security challenges. There are usually multiple applications running in parallel and communicating with each other, the RTOS and multiple space vehicle components (such as instruments, core hardware subsystems, and hosted payloads). Developers must address each mission application's implementation decisions and security concerns. Some concerns, such as programming safety, may be common to all applications if they are using the same language and development environment. Others, however, may depend on application functionality. One example is handling of communication protocols, performed by the command and data handling (C&DH) subsystem. Any processing of unvalidated input presents a possible attack surface and code that serializes or deserializes protocol traffic is especially vulnerable to such attacks and requires extra attention. Mission applications may also use a common middleware application programming interface (API) or shared code libraries, and these may also introduce unique security concerns that can affect every subsystem that uses them. Some space vehicles host multiple payloads, instruments, or subsystems with their own firmware, and the interactions between these parts of the architecture may provide adversaries ways to hide and additional weaknesses to leverage. Interfaces between separate parts of the system should be especially scrutinized, especially those using communication busses which may not have any security in their design (such as MIL-STD-1553).

F. Programming Language Selection

Systems programming languages typically provide access to low-level computing and hardware functionality to meet timing and resource management guarantees necessary for the performance of embedded systems, such as space vehicles. However, not all languages are created equal and some carry significant risk and flight software developers have recognized the need to consider safer languages for development. Snavely, Meyers, Inacio and Runyon [25] provide a perspective on choosing a language for flight software development, consider the history of safer languages for systems development (including C, C++, Ada, Java, and Rust) and detail an approach that leverages an ISO/IEC standard for avoiding vulnerabilities through selection of a safer language [26]. As Wheeler *et al.* point out, the aerospace industry does not seem to be moving to these newer languages *en masse* [16].

There are several features to consider in the selection of a programming language that provides the functionality necessary for systems programming, while providing various types of safety against common attack vectors. Memory safety has been the primary concern of many efforts over decades of computer science research. This includes both tracking and proper usage of memory allocation techniques, but also defensive measures that interface with hardware, such as memory addressing schemes (e.g. ASLR). Type safety is another area of consideration around the management and usage of data structures within programs. A well typed program only allows operations to occur that are a property of the relevant data, and the degree by which the language enforces these operations provides a way to characterize and understand risk. Some languages only support simple types built into the language (such as integers, floats, strings) while some allow for creation of algebraic data structures that can represent complex data as well as the operations that can be performed. If a language provides complex type safety functionality, the domain can be encoded into the type system and the language's features can prevent undefined or anomalous behavior.

Modern-day programming language design has provided numerous options to consider that can still provide systems programming features such as direct memory access and low-level control of hardware while avoiding unsafe behaviors, and in some cases guaranteeing that unsafe behavior can not be implemented at all. When considering undesirable behavior to eliminate, the biggest and most targeted area is memory safety. Recently the National Security Agency released official guidance recommending memory-safe languages [27]. The C language has well-documented flaws that allow for unsafe usage [28]. Numerous efforts exist to extend C with memory safety, including the CompCert C compiler [29] and Microsoft's Checked C project [30].

Rust is by far the most well known and mature language implementing memory safety as a core design principle, and is picking up steam in many systems-programming communities, including being used for portions of the Linux kernel (which is otherwise implemented in C) [31]. Other languages with memory safety include Ada SPARK [32], D [33], Nim [34], and Ivory [35]. There is existing work to extend Rust with formal methods and other high assurance techniques, including the Rust Formal Methods Interest Group [36], the High Assurance Rust project [37], and the Prusti formal verification project [38], among others. The application of these languages to the space domain is nascent and largely unexplored.

Additional features from the cutting edge of programming language theory provides some ways to safely reason about programs that, while not production-ready today, may make their way into systems languages in the future as additional ways to restrict unsafe behavior. One promising example is dependent type systems, which can add safety by embedding additional information about types into programs, allowing for better static analysis, as demonstrated in the Deputy system [39]. Another area are types systems based on linear logic, which can provide ways to reason about and control resource allocation [40].

IV. A Proposed Research Agenda

The following is a proposed research agenda for investigating how to develop cyber resilience in flight software. This agenda evaluates the proposed design considerations and outlines activities that can strengthen or add cyber resilience to various activities. These additions can improve security where it is weak, help gain a greater understanding of adversary attacks and capabilities against specific portions of flight software, and help evaluate the utility and effectiveness of cyber resilience-focused approaches.

A. Develop Cyber Resilience Requirements for Flight Software

Flight software development projects should have requirements dealing with needs related to cyber resilience, which can be verified, validated, implemented and tested alongside other mission requirements. This activity would consider a specific mission and develop a set of cyber resilience requirements tailored to that mission, as well as an implementation guide for how to do the same process for other projects, using lessons learned from the sample project. These requirements can address multiple cyber resilience goals across a broad swathe of flight software implementation areas, ensuring that developers mitigate possible vulnerabilities at the most appropriate parts of the overall process. Quality assurance processes can be leveraged as insertion points for implementation of these requirements. Quality-related software development processes can be modified with additional checks against commonly used adversary techniques and weaknesses in languages and frameworks. A framework such as SPARTA can be used to check coverage of adversary tactics and techniques. Coding guidelines and style guides can be modified to ensure that developers do not use language or tool features that are not secure or properly mitigated in the software architecture.

NIST has published cyber resilience guidelines that could be used as a starting point [4], and Bailey has produced a high level set of guidelines that can be expanded upon [9]. These guidelines should be converted into technical standards that apply to software development, procedures for testing software using adversarial methods, exercises and tabletops for evaluating processes, and reviews that developers can integrate into the existing flight software lifecycle. While this is the intention of the recently launched IEEE Standard Association's Standard for Space System Cybersecurity Working Group, an international effort to develop a technical standard for space system cybersecurity, there is much work to be done [8].

B. Investigate RTOS Security in the Context of Flight Software

The RTOS is the bedrock of flight software and deserves special scrutiny for security. This activity would perform detailed studies of RTOS security as it relates to programming language features, system design considerations, and adversary techniques specifically targeted against RTOS features such as boot loaders, low level memory access and device drivers. This would be done in the context of space vehicle development and the unique needs of space systems, since many RTOSes used for flight software are not custom-built for that use case. This would also consider existing security features of RTOS' and measure their effectiveness against attacks unique to the space domain, including such features as isolation and the usage of formal methods for checking specific segments of RTOS code. Different RTOSes have distinct features and security qualities, and an analysis of alternatives of existing solutions to identify gaps and understand common attacks that would be performed. Further work should consider restricting the capabilities of an RTOS to only serve the purpose of flight software, thereby reducing the attack surface.

C. Investigate Mission Application and Payload Security

Similarly to the RTOS investigation, mission applications, additional payloads, and the sections of the flight software above the RTOS should also be investigated for security in the context of space systems. This activity would perform detailed studies into application-level security between and within mission applications and payloads. This would include analyzing interactions between applications, communications, network and protocol handling, busses, interfacing with hardware, task management functions, scheduling functions, and other services provided by the flight software. Several parts of the application layer deserve special scrutiny and are called out separately in this agenda.

D. Perform Detailed Analysis of Command and Data Handling Subsystems

The command and data handling (C&DH) subsystem is possibly the most important subsystem within a flight software stack and deserves additional attention from a security and implementation perspective. This activity would analyze the functionality and implementation of the C&DH subsystem, including how commands are handled, command prioritization, how the C&DH subsystem interacts with other important subsystems, and how mission data is handled. Adversary attacks such as command intrusion are especially important and hardening the C&DH subsystem against these attacks should be investigated. Some missions have critical commanding that happens within the radio or RF subsystem. The critical commanding function's implementation and how it relates to the rest of the C&DH subsystem should be analyzed, as this presents an appealing target for adversaries. Another area of significant interest is the safe mode behavior of space vehicles, including how it is initiated, handled by the C&DH subsystem, and how transitions are handled between safe mode and any other operating modes of the space vehicle.

E. Perform Detailed Analysis of Guidance, Navigation and Control Subsystems

Another important subsystem that adversaries may seek to influence or modify is the guidance, navigation and control (GNC) subsystem. This activity would analyze the functionality and implementation of the GNC subsystem in a threat context. One style of developing a GNC subsystem is developing a model-based design using tools like Matlab and Simulink, where the necessary guidance algorithms are implemented in a high level language then transpiled into C code that is then embedded into the flight software. A security audit and vulnerability analysis of the toolchain and GNC implementation would be performed, to understand what security features are present in the host language and toolchain, what kind of C code the transpiler generates, and if any code analysis is done on the original code or final generated code. Alternatives that use safe programming languages would be investigated or proposed in this activity. In addition, the performance of the guidance algorithms themselves may be susceptible to perturbation or modification by adversaries and should be evaluated, and GNC subsystem connections to other subsystems should be inspected. Like the quality assurance processes described before, safety is an integral feature of space vehicle guidance, but defense against adversary attack and integrity of guidance data is likely not.

F. Perform Detailed Analysis of Autonomy Subsystems and Processes

Flight software implements many autonomy-related features of modern space vehicles. This activity would investigate the security landscape around the development of autonomy rules and the implementation of the flight software subsystems that store, manage and execute those rules. Autonomy in space is getting more complex, with robotic rendevous and missions with little to no contact with ground-side controllers planned [41]. Adversaries will seek to modify or influence autonomy to their own ends, so the tactics and techniques available must be analyzed and mitigations developed. Similar work exists for autonomous underwater vehicles and could be leveraged and extended into the space domain [42].

G. Develop a Deeper Understanding of the Flight Software Threat Landscape

Frameworks such as SPARTA [7] and TREKS [43] provide a way to understand the currently known landscape of adversary tactics and techniques available to attack flight software. However, this work is fairly recent and needs expansion and refinement in several ways. First, the full breadth of tactics and techniques are likely not captured by the current frameworks. Second, each existing technique does not have a full body of evidence showing how adversaries leverage those techniques, either theoretically or via real-world evidence. Third, protection measures and mitigations are not available for every technique, or sample implementations available to leverage for new systems under development. To provide better guidance on the breadth and depth of the threat landscape, activities should be stood up that explore these areas so that better information is available to flight software developers and testers. Comprehensive examples exist that demonstrate the usage of SPARTA for complex space system attacks, such as the "PCSpoof" attack on Time-Triggered Ethernet [44] [45] and an analysis of the demonstration of the takeover of an ESA satellite during the CYSAT conference [46] [47].

H. Investigate Adversarial Persistence and Exploitation Within Flight Software

As with any other type of operating system and computing environment, adversaries seek ways to gain a foothold, perform exploitation, maintain persistence, and perform lateral movement within the system and between systems. How adversaries perform these activities on ground-based systems is well-known, but performing them within a flight software architecture is different and not well explored in the open literature. This activity would seek to characterize, describe, and build proofs-of-concept for how adversaries can perform exploitation and persistence within the FSW environment, especially in the case of little to no command and control from the ground. FSW features exist that can be taken advantage of by malicious payloads for persistence (e.g. task scheduling routines) and due to FSW systems commonly lacking internal access controls or security-focused separation between processes, a payload may have multiple ways to find and exploit weaknesses in implementations that can be used for persistence, manipulation of internal data, and other cyber effects. An addition to this activity would consider how adversaries can leverage weaknesses in flight software to perform lateral movement within a single space vehicle and between space vehicles connected with cross-links, never requiring communications with the ground.

I. Develop a Methodology to Understand the Attack Surface of Flight Software

Understanding the attack surface available to an adversary is a fundamental part of engineering a software system that is resilient against cyber attack. Frameworks such as SPARTA are used to categorize and characterize the techniques necessary to perform a full attack, and what parts of the attack surface each technique uses. With this information, flight software developers can prioritize and begin mitigating these techniques. Many current efforts only focus on specific parts of the software used in a space system, such as the RTOS or specific mission applications. A comprehensive methodology that looks at the entire software stack present on a space vehicle should be developed and used to fully understand where adversaries may be able to take advantage of a system. This activity would seek to perform a set of these types of surveys to build a comprehensive methodology.

One way of characterizing the software stack is architecturally, starting with the lowest-level code and moving up to user-facing applications. A typical decomposition of these layers might look like this:

1) CPU running microcode
2) operating system kernel CPU/memory management
3) operating system kernel core services (task scheduler, process handling, etc.)
4) operating system kernel system call interface and device drivers
5) operating system userspace services (language runtimes, device handling)
6) application layer core services and API
7) mission-specific applications (possibly multiple layers)

Besides this layer cake, there's a breadth-wise relationship between multiple user-space applications running on the same OS, the interfaces between which depend on the system architecture. There is also a breadth-wise relationship between the CPU and other hardware components residing on the system, each of which may have interfaces with the operating system.

All these of these touchpoints are interfaces where two software components must communicate, be it by low level memory access, over a shared interface (bus or network), with a specific API, or a combination of all the above. Each interface is a seam in the system that, if used incorrectly, can cause undefined behavior or unknown state if the software component is not designed to handle the misusage. Several questions arise when considering how the flight software should securely operate.

1) Understanding the spread of interfaces across the entire system - how many are there, where are they in the system in relationship to each other?
2) What are the interfaces used to do? What components do they connect?
3) How are the interfaces developed? What language/protocol/schema is used to communicate using the interface?

Expanding this view to the entire system, an attacker can look at the combined set of attack surfaces and think about ways to cause effects on the behavior of the system to their desired end.

From an attacker point of view, the related questions become:

1) To produce a desired effect, what interfaces do I have to traverse?
2) Which interfaces do I need to use and are there ways to abuse their functionality?
3) Are there low-level design flaws in each interface on my desired path that I can utilize to produce the behavior I'm looking for that will allow me to move to the next step in my attack graph?

Current typical vulnerability analysis practices are empirical and search driven. Vulnerability analysts look for bugs or poor design decisions and then try to derive other impacts from them. They examine previous vulnerabilities for applicability to the system under test. While this is a useful process in terms of ensuring that a system is not vulnerable to specific types of flaws, this is not fundamentally driven by the design of the system and the mission it performs. Where the system has been historically secure or not secure does not necessarily correlate with where the system must be the most secure. Therefore, this approach only gets a system partway to being cyber resilient.

J. Investigate Implementation of Adversarial Testing for Flight Software

Simply performing analysis on source code to search for weaknesses is not enough to ensure that flight software is resistant against attack; live systems must be tested by developing appropriate testbeds and test harnesses, emulating adversary behavior and techniques, and attacking target systems *in situ*. This activity would develop methods and tools to attack running flight software, discover weaknesses and vulnerabilities, and measure their impact within the context of a mission. Various testbeds would be developed (or existing testbeds modified) to allow for use of these for vulnerability analysis, red teaming, and penetration testing activities, across a spectrum of fidelity levels as needed to perform or fully understand the impact of adversary behavior on mission effectiveness. A methodology can then be developed that can use this adversarial information to inform security-based implementation guidance at various stages of the software development lifecycle - development, IV&V, various levels of testing and system checkout, and for active mitigation in operations.

K. Investigate Cyber Resilience of Software Within Complex Space Systems

Space systems have grown more complex than a simple single-board computer running a single flight software stack. Many space vehicles are being proposed, developed and built that use multiple computers, on-board networks, multiple communication architectures, and are parts of large space-based networks. The Artemis cislunar architecture is an example where multiple, complex, and in some cases crewed space vehicles will be in operation and will have many embedded computing and networking components. These complex designs require a different security and cyber resilience analysis process. This activity would investigate and develop cyber resilience and secure systems engineering guidance for the software implementations within complex space systems, space stations, lunar bases and other large-scale craft where there is a complex interplay of software, networking, and hardware. Security beyond the operating system itself, to the network layer, will be evaluated, given the existence of a cislunar ”internet” using delay-tolerant networking to communicate within the architecture and with Earth. Future missions will extend this architecture to Mars and beyond. Cyber defenders will not be physically present to access these directly, and communication times with Earth will be longer, so autonomous cyber defense, security orchestration, and rapid recovery technology will be important to ensure cyber resilience and mission resilience. The techniques and technology necessary to implement these cyber resilience needs in space systems doesn't exist today and must be developed.

L. Develop Prototype Secure-by-Design Flight Software Architecture

We believe developers must take a threat-informed look at flight software architectures in light of the many issues raised in this paper and elsewhere. This activity should seek to address security needs by building a secure-by-design flight software architecture and implementing many of the security enhancements mentioned elsewhere. Formal methods and verification would be integral to the design, to guarantee correctness, ensure architecture requirements are validated, and verify behavior of the entire stack top to bottom. Techniques such as isolation, secure input handling, and memory safety would be fundamental to the design. Safe programming languages would be used throughout to implement all subsystems. Resilient approaches to core operating system functionality such as scheduling, task management, protocol handling, hardware interfaces, and interprocess communication should be utilized. The role of the RTOS should be scrutinized and alternatives to the current style of flight software architecture explored. Implementing a security-first architecture would allow for comparative analysis to be performed against older flight software systems so that risks can be properly understood and measured. This information would then be used to inform tradeoffs for future systems.

V. Conclusion

As space vehicles get more complex and more software driven, and determined adversaries increase efforts to cause harmful effects, to the need for better cyber resilience in their software design and implementation becomes more critical. Flight software must be actively redesigned to mitigate and neutralize adversary attacks and reduce their impact on the mission.

Current flight software quality work focuses on mission assurance and protection from loss, not on defending against adversary attack. When security is considered for flight software, most work today focuses on empirical vulnerability analysis, formal verification, and secure system models such as isolation. We believe this does not go far enough and that a fresh look at vulnerability analysis methods, software architectures, and programming languages is required.

This paper lays out a research agenda that addresses risks from cyber adversaries in both design and implementation and bring secure, cyber resilient software engineering methods firmly into space vehicle flight software development.

References

[1] The White House, “United States Space Priorities Framework,” Dec. 2021. [Online]. Available: https://www.whitehouse.gov/wp-content/uploads/2021/12/united-states-space-priorities-framework-_-december-1-2021.pdf

[2] 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

[3] Office of Space Commerce, “President Signs Space Cybersecurity Policy Directive,” Sep. 2020. [Online]. Available: https://www.space.commerce.gov/president-signs-space-cybersecurity-policy-directive/

[4] 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, Dec. 2021. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-160v2r1.pdf

[5] 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, Sep. 2018. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2018-5275

[6] B. Bailey, “ESTABLISHING SPACE CYBERSECURITY POLICY, STANDARDS, AND RISK MANAGEMENT PRACTICES,” Oct. 2020. [Online]. Available: https://aerospace.org/sites/default/files/2020-10/Bailey%20SPD5_20201010%20V2_formatted.pdf

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

[8] G. Falco, W. C. Henry, M. Aliberti, B. Bailey, M. Bailly, S. Bonnart, G. D. Rossi, N. Boschetti, N. G. Gordon, A. B. Byerly, D. Marsili, M. Bottarelli, G. Epiphaniou, D. Greaves, C. Maple, J. Brule, N. Tsamis, A. Carlo, M. Fetrow, J. D. Trujillo, D. Floreani, B. Jackson, R. Keen, S. Larson, and A. Viswanathan, “An International Technical Standard for Commercial Space System Cybersecurity - A Call to Action.”

[9] B. Bailey, “Protecting Space Systems from Cyber Attack,” May 2022. [Online]. Available: https://aerospacecorp.medium.com/protecting-space-systems-from-cyber-attack-3db773aff368

[10] NASA, “NASA Software Engineering Requirements.” [Online]. Available: https://nodis3.gsfc.nasa.gov/displayDir.cfm?Internal_ID=N_PR_7150_002D_&page_name=AppendixD

[11] “ECSS-E-ST-40C – Software (6 March 2009) | European Cooperation for Space Standardization.” [Online]. Available: https://ecss.nl/standard/ecss-e-st-40c-software-general-requirements/

[12] California Institute of Technology, "F' Code and Style Guidelines." [Online]. Available: https://nasa.github.io/fprime/UsersGuide/dev/code-style.html

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

[14] G. J. Holzmann, "Landing a Spacecraft on Mars," *IEEE Software*, vol. 30, no. 2, pp. 83–86, Mar. 2013, conference Name: IEEE Software.

[15] 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, Nov. 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

[16] 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, Sep. 2018. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2018-5220

[17] M. Asbury, "Jon McBride Software Testing and Research (JSTAR)," Mar. 2015. [Online]. Available: http://www.nasa.gov/centers/ivv/jstar/JSTAR.html

[18] B. Bailey, "NASA IV&V's Cyber Range for Space Systems," Feb. 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

[19] J. Backes, D. Cofer, S. Miller, and M. W. Whalen, "Requirements Analysis of a Quad-Redundant Flight Control System," in *NASA Formal Methods*, K. Havelund, G. Holzmann, and R. Joshi, Eds. Cham: Springer International Publishing, 2015, vol. 9058, pp. 82–96, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-319-17524-9_7

[20] M. Munoz, "Space systems modeling using the Architecture Analysis & Design Language (AADL)," in *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Nov. 2013, pp. 97–98.

[21] seL4 Project, "The seL4® Microkernel." [Online]. Available: https://sel4.systems/

[22] J. Windsor, K. Eckstein, P. Mendham, and T. Pareaud, "Time and space partitioning security components for spacecraft flight software," in *2011 IEEE/AIAA 30th Digital Avionics Systems Conference*. Seattle, WA, USA: IEEE, Oct. 2011, pp. 8A5–1–8A5–14. [Online]. Available: http://ieeexplore.ieee.org/document/6096140/

[23] 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, Sep. 2017, publisher: Royal Society. [Online]. Available: https://royalsocietypublishing.org/doi/10.1098/rsta.2015.0401

[24] 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, Sep. 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3230627

[25] 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, Sep. 2018. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2018-5397

[26] ISO/IEC, "ISO/IEC TR 24772-1:2019, Programming languages — Guidance to avoiding vulnerabilities in programming languages." [Online]. Available: https://www.iso.org/standard/71091.html

[27] National Security Agency, "Cybersecurity Information Sheet - Software Memory Safety," Nov. 2022. [Online]. Available: https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF

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

[29] X. Leroy, "CompCert." [Online]. Available: https://compcert.org/

[30] "Checked C." [Online]. Available: https://www.microsoft.com/en-us/research/project/checked-c/

[31] Internet Security Research Group, "Memory Safety for the World's Largest Software Project," Jun. 2022. [Online]. Available: https://www.memorysafety.org/blog/memory-safety-in-linux-kernel/

[32] AdaCore, "About SPARK." [Online]. Available: https://www.adacore.com/about-spark

[33] D Language Foundation, "D Programming Language." [Online]. Available: https://dlang.org/

[34] "Nim Programming Language." [Online]. Available: https://nim-lang.org/

[35] Galois Inc., "The Ivory Language." [Online]. Available: https://ivorylang.org/

[36] "Rust Formal Methods Interest Group." [Online]. Available: https://rust-formal-methods.github.io/tools.html

[37] T. Ballo, M. Ballo, and A. James, "High Assurance Rust: Developing Secure and Robust Software," 2022. [Online]. Available: https://highassurance.rs

[38] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers, "The Prusti Project: Formal Verification for Rust," in *NASA Formal Methods*, ser. Lecture Notes in Computer Science, J. V. Deshmukh, K. Havelund, and I. Perez, Eds. Cham: Springer International Publishing, 2022, pp. 88–108.

[39] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula, "Dependent Types for Low-Level Programming," in *Programming Languages and Systems*, ser. Lecture Notes in Computer Science, R. De Nicola, Ed. Berlin, Heidelberg: Springer, 2007, pp. 520–535.

[40] M. Triay, "Rust and Linear types: a short guide," Mar. 2023. [Online]. Available: https://medium.com/@martriay/rust-and-linear-types-a-short-guide-4845e9f1bb8f

[41] A. Banerjee, M. Mukherjee, S. Satpute, and G. Nikolakopoulos, "Resiliency in Space Autonomy: a Review," *Current Robotics Reports*, vol. 4, no. 1, pp. 1–12, Mar. 2023. [Online]. Available: https://doi.org/10.1007/s43154-023-00097-w

[42] K. Siil, A. Rubin, M. Elder, A. Dahbura, M. Green, and L. Watkins, "Mission Assurance for Autonomous Undersea Vehicles," in *2020 IEEE Security and Privacy Workshops (SPW)*, May 2020, pp. 244–249.

[43] J. G. Oakley, "TREKS Cybersecurity Framework," 2023. [Online]. Available: https://treksframework.org/

[44] A. Loveless, L. T. X. Phan, R. Dreslinski, and B. Kasikci, "PCspooF: Compromising the Safety of Time-Triggered Ethernet." IEEE Computer Society, Oct. 2022, pp. 3193–3208. [Online]. Available: https://www.computer.org/csdl/proceedings-article/sp/2023/933600a572/1He7YmWugq4

[45] B. Bailey, A. B. Byerly, G. Falco, t. dafoe, and B. Roeher, "Introducing SPARTA using PCSpooF: Cyber Security for Space Missions," Dec. 2022. [Online]. Available: https://aerospacecorp.medium.com/sparta-cyber-security-for-space-missions-4876f789e41c

[46] A. Martin, "Hackers to show they can take over a European Space Agency satellite," Apr. 2023. [Online]. Available: https://therecord.media/space-cybersecurity-satellite-hacked-esa-thales

[47] 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