EasyChair Preprint

№ 10604

A Research Agenda for Space Flight Software Security

James Curbo and Gregory Falco

July 21, 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 any space vehicle's successful mission operation. The reliability of flight software is not a new topic and has been studied extensively for the past decades through the lens of 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, a related but separate topic. 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 of flight software from future threats.

II. Cyber Resilience and Flight Software

A. Defining Flight Software

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

B. Why is Cyber Resilience Necessary?

Space missions require highly reliable hardware and software components, therefore value is assigned to proven systems that have flown on successful missions (so-called "heritage" components). Historically, spacecraft have not been a prominent target of cyber attack, so their developers have not built them with cyber resilience in mind. 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 spacecraft 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

Flight software is a prime target for attack by malicious parties. Because of the central nature of the flight software for operation of the spacecraft, access to it can lead to partial or total control of spacecraft 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 spacecraft users access them remotely, a secure ground system does not necessarily mean a spacecraft is secure as well. In addition, the growing complexity of software architectures on spacecraft, 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 spacecraft's systems and introduces a supply-chain aspect to securing these systems.

There is a lack of open source literature about cyber attacks against spacecraft. The Space Attack Research & Tactic Analysis (SPARTA) project [1] was recently started to document and categorize threats against spacecraft but is in its early stages and needs validation and extension.

D. Moving Towards Space 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 [2], 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 [3]

Bailey [4] lays out a set of cyber resilience principles that can be applied to the development of spacecraft. 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. Prior Art

There are many existing techniques, processes and methodologies for developing flight software. They exist to increase quality, handle mission-specific needs, ensure reliable spacecraft operation, and ensure the software produced meets user requirements. However, most flight software development processes do not address cyber threats or consider adversarial attacks. This section highlights existing work and where that work falls short of mitigating or preventing cyber attacks.

A. Quality Assurance Processes

Many flight software projects have associated quality assurance standards, processes, and associated coding guidelines. Mission designers use these guidelines to ensure that the flight software meets the requirements of the mission operators and will execute according to its intended design. These standards can also include safety-related guidelines, if necessary for the operation of the spacecraft, for example in human spaceflight missions. These guidelines are used to define testing and independent verification and validation (IV&V) processes used by developers of flight software. Examples of standards in this area include NASA's Software Engineering Requirements [5], the European Space Agency (ESA) code standards [6], the Jet Propulsion Laboratory (JPL) F' flight software system [7], and JPL's standards for using the C programming language [8]. Holzmann provides a detailed case study of software quality in the Mars Science Laboratory mission [9] and highlights the need for reliable code and detailed software engineering processes.

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 attack or related security concerns. Also, the pace of the development of cyber threats often outpaces the development of complex systems such as spacecraft, so even with the best security requirements up front, the system may not be fully secure. These reasons make quality standards alone insufficient, but not unnecessary, for increasing 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 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.* [10] present a detailed discussion of the use of static code analysis in flight software. 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. [11]

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. Formal Methods

Various methods exist to leverage formal, rigorous verification and validation methods to automatically or semi-automatically prove the correctness of software components in flight software. These methods vary, but all focus on the use of formal specifications to drive correctness and eliminate unsafe or undesired behavior from the root of the implementation.

1) Formal Verification of Architectures and Model-Based Design: One method, requirements verification, uses a set of formal requirements to drive a verification framework that is embedded in the software code base. This framework drives a set of unit and integration tests that are used to report the compliance of specific software components against the necessary requirements. One example of this method was the verification of the Parker Solar Probe flight software via formal requirements. [12]

Another method uses a model-based design philosophy to develop, test, and validate the behavior of software components through simulation. In this approach, the environment that the software must interact with is extensively modeled and used to generate software code as well as drive simulation inputs for testing. This approach cuts out a step between design and implementation as the implementation becomes the design (also called an "executable specification"). Developers often use this approach for the guidance, navigation & control (GN&C) subsystem, since it must implement control system algorithms that must work in real-time. An example of this approach is the GN&C flight software for the Orion Crew Exploration Vehicle. [13]

These methods are useful for constraining a flight software system to a limited set of behaviors based on formal specifications (executable or not) and setting up comprehensive testing and evaluation frameworks to ensure compliance. However, from a cyber-resilience point of view, they still fall short in several ways. Eliminating undesired behavior may reduce the attack surface available for adversaries, but there are numerous examples in cybersecurity literature of adversaries using legitimate methods to perform attacks. Furthermore, requirements verification activities that set up unit test driven frameworks still operate on an empirical basis, starting as lists of desired behavior and then reporting on the compliance or non-compliance of specific software components but not preventing unsafe behavior. Finally, many current implementations of these methods use unsafe programming languages, and even if unsafe features of the languages are forbidden, undefined behavior may still be an issue.

2) Formal Verification of Software: Formal verification techniques can also be used by developers within programming languages directly, to prove the correctness of implemented algorithms. Formal verification techniques are often used for code that must be extremely robust and failure-proof, such as cryptographic algorithms or control systems. Various techniques and tools exist to perform formal verification depending on the language and environment in use.

Real-time operating systems, being low-level interfaces to the hardware and implementing core processing functions, are often targets of formal verification efforts. The FreeRTOS Project [14], which produces a commonly used operating system in flight software stacks, has used formal verification techniques to analyze critical sections of the FreeRTOS code

base, including analyzing network packet handling code using a C model checker to look for memory safety and other issues. [15]

While model checking can find many issues in languages such as C that are popular for systems programming (and flight software), it is still an empirical process working within a set of programming language design decisions with inherent flaws. Because of the exhaustive search necessary for full model checking, it can be expensive in time and space to check a large code base. The state explosion problem is a well known issue in the development and implementation of model checkers, with various optimizations and workarounds available. [16]

D. Programming Language Selection

C and C++ are by far the most popular languages for developing low level embedded code such as flight software given its powerful low-level functionality. However, the C language has flaws that allow for unsafe usage. These flaws are well documented [17] and while workarounds exist, such as restricted compilers and restricted subsets of the language, C is still used, even for new projects.

Flight software developers have recognized the need to consider safer languages for development. Snavely, Meyers, Inacio and Runyon [18] 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. [19]

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. [20] Numerous efforts exist to extend C with memory safety, including the CompCert C compiler [21] and Microsoft's Checked C project [22]. 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). [23] Other languages with memory safety include the venerable Ada and newer languages such as D [24], Nim [25], and Ivory. [26]

E. Architectural Design

While use of safe programming languages is one way to constrain behavior, designing software architectures with security-specific features in mind is another way to limit unsafe and unwanted system behavior. 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 spacecraft design. [27] [28]

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 [29]. 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. [30]

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 [31] describes the ways formal methods were used in application development during the project, and Klein (et al) [32] detail the use of the seL4 microkernel and how it was used to derive security guarantees for the entire architecture.

IV. Required Security Augmentation of Current State

The need to create flight software systems that are resilient to cyber attack requires consideration of choices in design and implementation across multiple areas. This section highlights several areas where we deem current work insufficient to mitigate adversary cyber attacks and prevent production of unwanted or harmful mission effects. Developers must purposefully analyze flight software systems with the adversary's intentions and capabilities in mind, and develop methods and tools that respond to those attacks.

A. Complementing Quality Assurance with Cyber Resilience

As the flight software is a critical part of the overall system, reliability and quality are prioritized. Flight software, along with other parts of the space system, undergoes intense testing and must pass rigorous quality assurance processes to meet design goals and project milestones. However, these processes are not designed around evaluating and mitigating risks due to adversary cyber attack. These risks may overlap with quality risks, but adversaries can utilize legitimate parts of a system to pursue their goals, which a quality process will not recognize. Therefore, processes that analyze a system from an adversary's point of view are necessary to enable true cyber resilience. These processes must be complimentary to and considered with relevant quality processes to enable overall system resilience.

B. Understanding the Attack Surface of Flight Software

Understanding the attack surface available to 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 spacecraft is necessary to fully understand where adversaries may be able to take advantage of a system. This stack encompasses a wide variety of software components that may require specific approaches based on the technology used. A comprehensive survey gives flight software developers the means to prioritize mitigation activity across the entire system and is a critical component of the overall process.

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.

C. Security of the RTOS

RTOSs are foundational to all flight software functionality. Therefore, increasing the security of the RTOS layer is one way to increase the overall resilience of flight software. Because of the need for RTOSs to be lean, fast, and efficient, system developers use low-level systems programming languages such as C. However, C and similar languages have qualities that lead to unsafe design and so many tools have been developed to mitigate and remediate issues in RTOSs. These include static and dynamic code analysis methods, model checking, and implementation of critical portions of the RTOS in formally verified languages. RTOS developers should expand and standardize these practices within their development projects.

To improve upon these techniques, which are empirical in their coverage, developers should consider a mission-oriented point of view. The SPARTA framework gives us a way to analyze adversary techniques in the context of the RTOS. One method would be to consider each SPARTA technique, ask how an adversary would take advantage of that technique, and how a developer can mitigate the technique in the context of the RTOS. The results of that process will give developers a list of starting points for remediation work, security analysis, and further investigation.

D. Security of Mission Applications

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 spacecraft 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.

E. Formal Methods and Safe Programming Languages

Systems programming languages with access to low-level computing functionality are a necessity for real-time, embedded system development, such as that necessary to design spacecraft. Previous work on selection of programming languages for flight software (Snavely et.al) focused on understanding the features of languages, associated risks, and mitigations thereof, based on ISO/IEC 24772 ("Guidance to avoiding vulnerabilities in programming languages through language selection and use"). Previous efforts focused on constrained subsets of system-level languages (e.g. C or Ada). However, the growth of newer languages with built-in safe features, such as Rust, provides an opportunity to start from a more secure baseline, from a language perspective, and focus resilience efforts on other problems. In addition, the proliferation of formal methods-based toolkits and language support provides another way to add resilience to flight software systems. As Wheeler *et al.* point out [10], the aerospace industry does not seem to be moving to these newer languages *en masse*. However, combining this risk analysis with construction of a fundamentally new system architecture divorced from the RTOS/application split of the past provides an opportunity to address these risks in a foundational way, while increasing resilience and assurance of the entire system.

There are several types of safe language features to consider in the design of a programming language being selected to build performant, real-time, cyber-resilient software. 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 candidates to consider outside of the C/C++ hegemony 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. As discussed above, the most prominent example is Rust, but there are alternatives available. There is existing work to extend Rust with formal methods and other high assurance techniques, including the Rust Formal Methods Interest Group [33], the High Assurance Rust project [34], and the Prusti formal verification project [35], among others. In addition, the cutting edge of programming language theory provides some language features that, while not production-ready today, may make their way into systems languages in the future as additional ways to restrict unsafe behavior, such as dependent types and linear logic.

F. Reconsidering the Typical Flight Software Architecture

A more foundational approach to flight software security asks why the RTOS and mission applications are separate in the first place. This style of design philosophy considers the full range of functionality necessary in the mission software and eliminates artificial divisions between mission applications, middleware and the RTOS. Following typical systems engineering principles, designers would derive operational needs of the spacecraft from the mission objectives, and would then turn them into functional and technical requirements. Some of these requirements would be fulfilled by dedicated software interfacing with appropriate hardware. In theory, this should lead to a set of software subsystems more aligned to the various elements of the mission and eliminate unused or redundant software components. This also allows the addition of resilience against cyber attack as a fundamental operational need, which should then flow into the rest of the system design.

One example of a cyber resilience-oriented operational need is secure isolation, authentication, and authorization between various system components, labeled a 'zero-trust' system architecture. With this as a core design principle, developers can integrate zero-trust design patterns up front. For instance, in a zero-trust spacecraft architecture, components communicating over the system's main bus would require authorization and authentication by default. Many spacecraft today implement interfaces such as MIL-STD-1553 that do not require this and are susceptible to other attacks, such as bus takeover attacks.

V. Towards Cyber Resilient Flight Software

To create flight software that is secure and resistant to cyber attacks, mission designers must build a more robust architecture with cyber resilient principles in mind. This must incorporate all the previously defined design considerations, and there are likely others that need to be discovered and

incorporated as well. Designers must explicitly integrate these features into the architecture via specification and model-based design, and into the software development process by selection of safe programming languages and development approaches. Flight software developers must have the ability to leverage these techniques without requiring a computer science education. For some of the design considerations, this is a straightforward introduction of existing techniques. For others, additional research is necessary to make the techniques production-ready.

A. Design a Modular, Integrated Flight Software Architecture

Instead of the typical separation between RTOS and mission applications, which uses an underlying RTOS that may not be explicitly designed for spacecraft, a novel approach could break down the existing barriers and treat the full flight software stack as one system. Designing a modular architecture would involve decomposing the typical RTOS and application layer functions and recombining them into a more integrated architecture. The architecture would then leverage verification and model-based design techniques to prove correctness and map out all possible interfaces between system components. This would allow for better understanding of possible attack surfaces and identification of where the system may need additional security protections, at the fundamental requirements specification stage. Advanced security design elements such as zero-trust techniques, privilege separation, and isolation can be incorporated and enforced at the design level via the architecture verification process.

B. Use Safe Programming Languages

To eliminate undesired and unsafe behavior, developers should write the entire flight software architecture in a safe programming language. Memory safety is an absolute requirement for a secure system, and the availability and priority of other elements of safety (such as the needed degree of type safety) should be evaluated as well. For these purposes, Rust is likely the best candidate in terms of production readiness and ecosystem maturity, but developers should perform an analysis of alternatives of languages to understand risk, tradeoffs and development requirements.

C. Prove the Cyber Resilience of the Flight Software Architecture Using Formal Methods

Developers should incorporate rigorous development techniques using formal methods at all stages and layers of flight software development to verify and validate the cyber resilience properties of the entire software stack. A variety of techniques exist, and developers should perform an evaluation of which techniques are tractable and contribute most to validation of cyber resilience. Many of these techniques are already being used for validation and testing of fault tolerance and could be extended to cyber resilience.

A verified kernel of some type should form the basis of any new flight software system. Developers should verify additional components on top of the RTOS, including components such as file systems and device drivers. Other targets for verification include protocol implementations and internal communications systems, such as busses, point-to-point connections and internal networks. Finally, developers should formally verify mission applications comprising the core subsystems of the flight software.

Developers should also consider formal methods for understanding and constraining the behavior of applications. Many attacks rely on timing, memory positioning, or memory access, and these could be caught through proper modeling of the larger set of activities being performed by multiple components of the system. Similar techniques have been used in the past for fault tolerance and quality related to task execution and system behavior and have prevented issues that are difficult to reason about, such as those related to concurrency.

At the code level, developers should increase the use of static code analyzers and model checkers to the maximum amount possible throughout the system's code base. Additional, newer code-checking techniques, such as symbolic execution and dynamic code analysis techniques, should be investigated and built into flight software development processes where applicable and feasible.

D. Verify Resistance from Threats by Using Adversarial Testing Techniques

In addition to formally modeling and verifying the proper behavior of the flight software architecture, much work could be done in modeling, analyzing, and representing possible cyber attacks. Efforts such as SPARTA are starting to quantify possible techniques used by adversaries. Security researchers should do more research to discover new techniques or expand current techniques documented in SPARTA. In addition, each technique within SPARTA needs to be modeled, developed, tested and proven in a wide variety of mission scenarios and environments. Formal techniques for modeling and testing these techniques should be developed and those test cases used as input to the formal specification of flight software architectures. At a system level, developers should use active adversarial testing techniques such as penetration testing and red/purple teaming exercises more often and at multiple stages of the systems engineering lifecycle to validate the cyber resilience of the flight software architecture. Early stage adversarial testing can provide feedback to the developers so changes can get implemented early on, while later stage testing can help with risk management and validation of proper system operation in specific mission scenarios.

VI. A Proposed Research Agenda

The following is a proposed research agenda for investigating how to develop cyber resilient flight software. This agenda evaluates the proposed design considerations, outlines activities that can discover additional elements of architecture design that may be necessary and builds an understanding of what additional techniques of secure software design and formal methods developers of flight software can leverage.

A. Develop Cyber Resilience Guidelines for Flight Software Systems Engineering

Flight software development projects should have cyber resilience guidelines added to their systems engineering processes and documentation. This research activity would consider a specific mission and develop a set of cyber resilience guidelines tailored to that mission, and an implementation guide for other projects using the lessons learned from the guideline development process. These guidelines can be added to the appropriate stages of flight software development to ensure developers mitigate possible vulnerabilities at the best times. NIST has published cyber resilience guidelines that could be used as a starting point [2], and Bailey has produced a high level set of guidelines that can be expanded upon. [4] 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 [36].

B. Modify Quality Assurance Processes for Security

Existing quality assurance processes are ideal places for adding cyber resilience because of their thoroughness, scale and widespread nature in spacecraft development. The mindset that drives detailed, impactful quality assurance can also drive the same qualities for cyber resilience. This activity would take an existing software engineering process and add additional checks to account for cyber resilience. Technical software development processes, such as those that ensure code quality, 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.

C. Investigate RTOS Cybersecurity

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 work would also consider existing security features of RTOS' and measure their effectiveness against attacks, 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 attacksthat 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.

D. Investigate Mission Application Cybersecurity

As with the RTOS, developers should examine the mission applications that carry out the majority of spacecraft functionality. This activity would perform detailed studies into application-level security as currently implemented in popular flight software. This would include an analysis of many of the operating system functions implemented and their security postures, including interprocess communication, inter-application system functions, networking and protocol handling, and interfacing with the RTOS and any operating system application layer, if present.

E. Perform Detailed Analysis of Command and Data Handling Subsystem

Several mission application subsystems deserve additional attention and the most important one is the command and data handling (C&DH) subsystem. 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 spacecraft, 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 spacecraft.

F. Perform Detailed Analysis of Guidance, Navigation and Control Subsystem

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. 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 would be evaluated, plus GNC

subsystem connections to other subsystems. Like the quality assurance processes described before, safety is an integral feature of spacecraft guidance, but defense against adversary attack and integrity of guidance data is likely not.

G. Develop Prototype Cyber Resilient Security-Focused Flight Software Architecture

We believe developers must take another look at flight software architectures, in light of many of the issues raised in this paper. This activity would outline a new flight software design that prioritizes security requirements. 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 would be utilized.

H. Investigate Cyber Resilience of Complex Space Systems

Space systems are growing more complex than a simple single-board computer running a single flight software stack. Many spacecraft 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 spacecraft 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 than what is needed for a single spacecraft. This activity would investigate and develop cyber resilience and secure systems engineering guidance for complex space systems, space stations, lunar bases and other large-scale craft. Security beyond the operating system itself 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.

I. Investigate Adversary Attacks on Autonomy Implemented in Flight Software

Flight software implements many autonomy-related features of modern spacecraft. 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. Adversaries will seek to modify or influence autonomy to their own ends, so the tactics and techniques available must be analyzed.

J. Perform Multiple Fidelity Level Vulnerability Analysis of Flight Software

Simply performing code analysis on source code is not enough to ensure that flight software is resistant against attack; live systems must be tested by attacking them *in situ*. This activity would develop methods and tools to attack flight software, discover vulnerabilities, and measure their impact at various stages of the software development lifecycle. Various testbeds would be developed (or existing testbeds modified) to allow for adversarial vulnerability analysis, red teaming, and penetration testing activities using known adversary tactics and techniques.

VII. Conclusion

As spacecraft 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.

The typical mission software design of applications running on an RTOS layer not designed for spacecraft may provide opportunities for adversaries to attack. Therefore, we see a need for a tightly integrated application/RTOS stack that can address the limitations of previous approaches. Understanding the attack surface of the entire software system, most importantly how adversaries may use the interfaces between software components, is necessary to mitigate risks from adversary attack.

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 spacecraft flight software development.

References

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

[2] 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

[3] G. Falco, W. Henry, M. Aliberti, B. Bailey, M. Bailly, S. Bonnart, N. Boschetti, M. Bottarelli, A. Byerly, J. Brule *et al.*, "An international technical standard for commercial space system cybersecurity-a call to action," in *ASCEND 2022*, 2022, p. 4302.

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

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

[6] "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/

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

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

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

[10] 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

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

[12] S. Jacobs and K. A. Wortman, "Solar Probe Plus Spacecraft Flight Software requirements verification test framework," in *2016 IEEE Aerospace Conference*, Mar. 2016, pp. 1–8.

[13] M. Jackson and J. Henry, "ORION GN&C MODEL BASED DEVELOPMENT: EXPERIENCE AND LESSONS LEARNED," in *AIAA Guidance, Navigation, and Control Conference*. Minneapolis, Minnesota: American Institute of Aeronautics and Astronautics, Aug. 2012. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2012-5036

[14] "FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions." [Online]. Available: https://www.freertos.org/index.html

[15] "The CBMC Homepage." [Online]. Available: https://www.cprover.org/cbmc/

[16] T. Do, A. C. M. Fong, and R. Pears, "HOW EFFECTIVE IS MODEL CHECKING IN PRACTICE?" in *Proceedings of the 6th International Conference on Evaluation of Novel Approaches to Software Engineering*. Beijing, China: SCITEPRESS - Science and Technology Publications, 2011, pp. 239–244. [Online]. Available: https://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0003467402390244

[17] 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

[18] 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

[19] 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

[20] 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

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

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

[23] 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/

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

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

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

[27] 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

[28] 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.

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

[30] 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/

[31] 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

[32] 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

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

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

[35] 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.

[36] "Standard for space system cybersecurity." [Online]. Available: https://standards.ieee.org/ieee/3349/11182/