Project Report
LSP-249

Guidelines for Secure Small Satellite Design and Implementation: FY18 Cyber Security Line-Supported Program

K.W. Ingols
R.W. Skowyra

February 6, 2019

Lincoln Laboratory
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LEXINGTON, MASSACHUSETTS

This material is based upon work supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001.

This material is based upon work supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Assistant Secretary of Defense for Research and Engineering.

Massachusetts Institute of Technology
Lincoln Laboratory

Guidelines for Secure Small Satellite Design and Implementation: FY18 Cyber Security Line-Supported Program

K.W. Ingols
R.W. Skowyra

Project Report LSP-249

February 6, 2019

Lexington Massachusetts

This page intentionally left blank.

ABSTRACT

We are on the cusp of a computational renaissance in space, and we should not bring past terrestrial missteps along. Commercial off-the-shelf (COTS) processors—much more powerful than traditional rad-hard devices—are increasingly used in a variety of low-altitude, short-duration CubeSat class missions. With this new-found headroom, the incessant drumbeat of "faster, cheaper, faster, cheaper" leads a familiar march towards Linux and a menagerie of existing software packages, each more bloated and challenging to secure than the last.

Lincoln Laboratory has started a pilot effort to design and prototype an exemplar secure satellite processing platform, initially geared toward CubeSats but with a clear path to larger missions and future high performance rad-hard processors. The goal is to provide engineers a secure "grab-and-go" architecture that doesn't unduly hamstring aggressive build timelines yet still provides a foundation of security that can serve adopting systems well, as well as future systems derived from them.

This document lays out the problem space for cybersecurity in this domain, derives design guidelines for future secure space systems, proposes an exemplar architecture that implements the guidelines, and provides a solid starting point for near-term and future satellite processing.

This page intentionally left blank.

ACKNOWLEDGMENTS

We gratefully acknowledge the alert editing and patient feedback provided by Jeff Brandon, John Grimes, Roger Khazan, and David Wilson, as well as helpful discussions with Andy Bentley, Nancy Crabtree, Eric Koziel, and Mike Zhivich.

This page intentionally left blank.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Page

LIST OF FIGURES

This page intentionally left blank.

LIST OF TABLES

This page intentionally left blank.

1. INTRODUCTION

Satellite systems provide a great many services to everything that lives and works beneath them. Some extend our understanding of the universe around us, voyaging to the edge of our solar system and beyond. Some provide valuable data about our own planet, serving civilian, intelligence, and military purposes. Some provide communications capabilities, both broadcast and point-to-point. Some are commercial, and some are government-owned. We almost take them for granted; it is increasingly difficult to remember life before the 32 satellites of the Global Positioning System (GPS) changed the way we navigate. Space systems are now part of our critical infrastructure.

For all their value, satellites come at a high cost. They must operate unattended in a harsh and unforgiving environment for extended periods of time. The European Space Agency's Envisat Earth observation satellite, for example, weighs more than eight tons and cost almost $3 billion to build. Getting the satellite from the ground to space is not cheap, either; launching one kilogram to low Earth orbit (LEO) costs several thousand dollars at the low end, and launching to geosynchronous orbit (GEO) is far more expensive. Clearly the operational utility of these satellites must be tremendous to justify such costs.

Not every satellite needs to weigh eight tons and cost $3 billion. Technological advancements are continually increasing what can be done with smaller satellites, and many missions may find that a smaller system or constellation is a more cost-effective method of meeting the mission need. A surprising amount of functionality can be packed into those small form factors. TacSat-4, for example, demonstrates smaller communications satellite technologies [4]. On the smaller side, the tiny "Dove" satellites developed by Planet, shown in Figure 1, will supply overhead imagery for commercial purposes in a tiny launch envelope—on the order of 10 lbs each. It was not that long ago that overhead imagery of this sort would include special reentry vehicles to deliver film canisters from the satellite back to the ground for development.

The most common small satellites are CubeSats. CubeSats are measured in industry standard units that are $10\times10\times10$ cm^3 in size and weigh roughly 1 kg — a cube called simply a "U." Common CubeSat sizes include 1U, 3U ($10 \times 10 \times 30$ cm), 6U ($10 \times 20 \times 30$ cm), 12U, and 27U. Developers are working to make ever more agile and composable development strategies to streamline concept-to-flight, e.g., [5, 6]. Launch services have evolved to support easy integration and deployment of standard-sized CubeSats, where rockets carrying multiple satellites each drive down launch costs by a substantial factor [7].

Due to their increasing popularity, a variety of commercial off-the-shelf (COTS) vendors (e.g., [8]) now sell components suitable for use on small satellite programs — a departure from the more bespoke nature of large satellites. Readily available COTS parts enable more aggressive acquisition timelines and budgets. The accelerated and leaner development environment leaves less breathing room for development that does not directly benefit reliability (to environmental threats) or functionality. However, as these smaller satellites become increasingly operationally relevant, they also become increasingly attractive targets to adversaries who may wish to deny satellite owners unfettered access to the capabilities they provide.

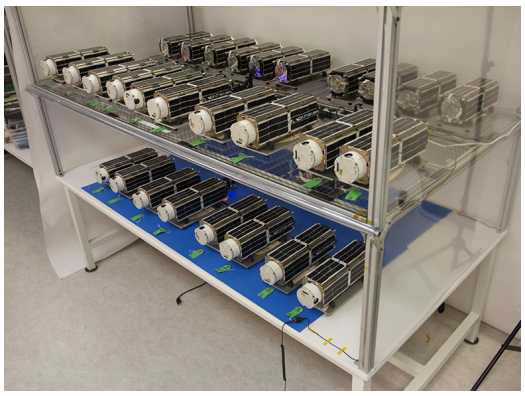

Figure 1. "Flock" of Planet "dove" satellites.

Securing small satellite platforms against cyber attack by these adversaries poses challenges unique to the space domain. Once in orbit, software updates to the satellite (both security-related and otherwise) become more difficult and expensive. Any update must be highly assured with respect to software bugs. A crashed process can have dangerous physical effects if it manages a hardware device, such as causing an orientation failure that results in a thermal imbalance or inability to charge via solar panels. Should control of the satellite be lost due to a cyber attack, the lack of physical access to the system's hardware makes it much more difficult for legitimate operators to reassert control than in an enterprise environment. Finally, the physical constraints of the space environment make it very difficult to recover from the result of a successful cyber attack, even if control can be regained. For example, if an attacker can cause orbit changes via firing of thrusters, that satellite may have insufficient fuel to recover even if the legitimate ground station is able to reassert control.

1.1 CONTRIBUTIONS OF THIS REPORT

This document contributes to the solution space by positing a series of security-relevant design guidelines for satellite systems, an analysis of the challenges in securely implementing these guidelines, and a reference architecture demonstrating the feasibility of securely applying them to a representative small satellite. The design guidelines often go beyond what may at first appear as security guidance (e.g., compliance with security controls). They instead focus on aspects of system design that can make a system foundationally securable, in contrast to traditional approaches where security is added on to a pre-existing architecture.

We begin by discussing secure system design in general terms in Section 2 and satellite design in Section 3. In Section 4, we bring the two disciplines together and discuss secure satellite system design, reviewing related work and applying the design strategies from Section 2 to the space environment. The discussion leads to and motivates a small handful of threat scenarios in Section 5, which in turn drive the development of a set of secure satellite design guidelines in Section 6 that, if followed, will provide useful improvements to satellite designs from inception to

insertion. In Section 7, we analyze several of the challenges faced by system architects in securely implementing our design guidelines.

Our project seeks to realize the guidelines in at least one representative architecture. The initial architecture and some of the implementation challenges we encountered working to instantiate it appear in Appendix A.

This page intentionally left blank.

2. OVERVIEW OF SECURE SYSTEM DESIGN

This section offers a whirlwind tour of related work on secure system design, with a heavy bias towards cybersecurity. There is deliberately little to no tailoring to the space environment here. Instead, we talk generically about cyber protections here, generically about the space environment in Section 3, and bring the two aspects together in Section 4.

2.1 THREATS AND THREAT MODELS

The majority of threats that space systems face (space junk strikes, radiation, EMI, etc.) remain obedient to the laws of physics. The capabilities of an adversary or the environment can be reasonably understood, accounted for, bounded, and mitigated.

Unfortunately, the cyber threat is far more challenging to quantify. The "space of cyberspace" is not well understood, nor is a "physics" of cyberspace sufficiently developed. Researchers are still discovering entirely new classes of attacks, and systems have ample room to hide as yet undiscovered vulnerabilities.

Although it's tempting to view "cyberspace" as a completely man-made, unreal environment, it is nevertheless rooted in reality. However, the path from JavaScript to transistors is too complex to trace, and the abstractions built up to address the complexity do not often remain true in the face of adversarial activity. Despite lacking a hard physics, the cybersecurity community has nevertheless pursued several approaches to characterize the threat.

The Common Attack Pattern Enumeration and Classification (CAPEC) resource [9] provides a "comprehensive dictionary and classification taxonomy of known attacks", in theory providing a way to clearly name the slings and arrows that are hurled at computer systems. The dictionary enumerates hundreds of attack mechanisms, including attacks against networks ("258: Passively Sniffing and Capturing Application Code Bound for an Authorized Client During Dynamic Update"), applications ("190: Reverse Engineer an Executable to Expose Assumed Hidden Functionality or Content"), supply chains ("539: ASIC With Malicious Functionality"), humans ("163: Spear Phishing"), and facilities. The hierarchical structure of CAPEC helps to "future-proof" the dictionary somewhat by covering both the broad strokes of a potential adversary action (e.g., "525: Execute Code") and the narrow strokes of currently known tradecraft (e.g., "579: Replace Winlogon Helper DLL").

The Defense Science Board (DSB) opted to back away substantially from a catalog of *techniques* and instead categorize the adversary by a very coarse-grained view of available *resources* [1], dividing the threat into three broad categories, as shown in Figure 2:

- **Tier I–II:** independent actors with modest means that use existing public exploitation tools and tradecraft or develop modest tooling based on known vulnerabilities;
- **Tier III–IV:** criminal or state actors with substantial funding that can develop and use their own tools, often for financial gain or espionage; and

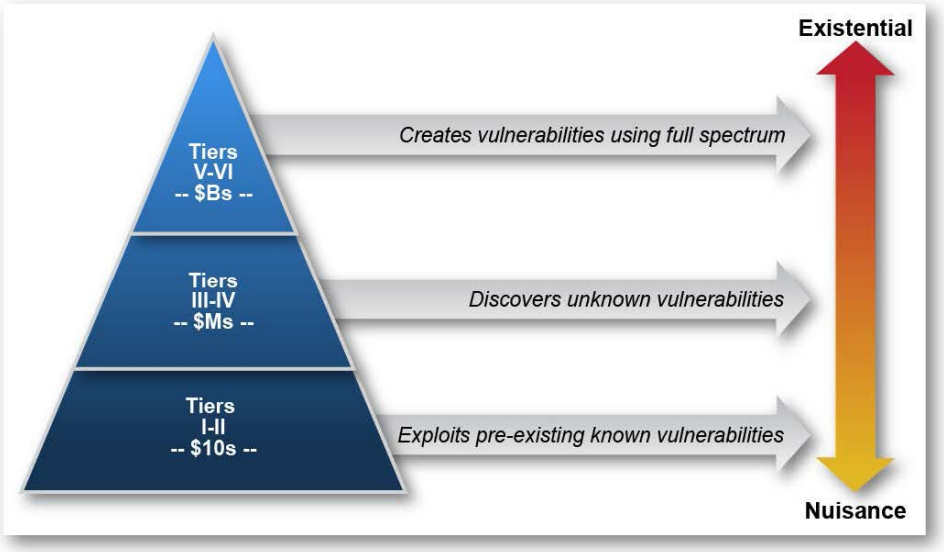

Figure 2. Cyber threat taxonomy [1].

- **Tier V–VI:** "full spectrum" state actors capable using cyber attacks in concert with other non-cyber military and intelligence techniques to carry out specific desired outcomes.

The first tier of adversary should be defeatable by proper application of existing defensive techniques—so-called "basic hygiene"—which has itself proven to be a challenge for many systems. The upper tiers rely on a rich stew of deterrence, deception, obfuscation, and research and development to survive that sea of troubles. The next section reviews schemes devised to identify and select appropriate mitigation techniques for a system.

2.2 RISK MANAGEMENT AND MITIGATION TECHNIQUES

This section reviews three approaches for selecting cyber defenses and assigning them to specific parts of a system for implementation. The Risk Management Framework (RMF) is worth careful review simply because it's the mechanism of choice for a variety of real systems, the organizations that fund their development, and the organizations that permit them to enter service. The MATRIX approach, described in Section 2.4, is a brainstorming technique that pulls in broad views of adversary techniques with an implicit consideration of adversary resources (a blend of CAPEC and DSB), adversary goals, and mitigation strategies. Finally, the STAMP suite of processes, described in Section 2.5, takes lessons from safety engineering and pivots them to security in a largely adversary-agnostic manner.

SC-24 FAIL IN KNOWN STATE

Control: The information system fails to a [*Assignment: organization-defined known-state*] for [*Assignment: organization-defined types of failures*] preserving [*Assignment: organization-defined system state information*] in failure.

Supplemental Guidance: Failure in a known state addresses security concerns in accordance with the mission/business needs of organizations. Failure in a known secure state helps to prevent the loss of confidentiality, integrity, or availability of information in the event of failures of organizational information systems or system components. Failure in a known safe state helps to prevent systems from failing to a state that may cause injury to individuals or destruction to property. Preserving information system state information facilitates system restart and return to the operational mode of organizations with less disruption of mission/business processes. Related controls: CP-2, CP-10, CP-12, SC-7, SC-22.

Figure 3. RMF sample control.

2.3 RISK MANAGEMENT FRAMEWORK

In the world of government security standards, the RMF is at present preeminent. The Risk Management Framework (RMF) is the current standard by which Air Force systems are secured, and the rest of the DoD is moving toward it as well [10]. The RMF process is based on the NIST 800-53 set of security controls [11]. Engineers review all 900+ controls and enhancements (sub-controls), select those appropriate to the system, tailor them to the system's needs, and use those selected controls to generate requirements for the acquisition process. An example control from 800-53 is presented in Figure 3.

RMF has been subject to its share of criticisms. One is that RMF predicates its control selection based on a "characterization" of the target system's confidentiality, integrity, and availability needs judged against mission impact [12], but it doesn't really speak to the system's deployed environment, mission lifetime, adversary capabilities and motivation, etc. This is largely a side effect of RMF's roots as a successor to the Federal Information Security Management Act (FISMA) checklists and its focus on enterprise computing, where aspects of the environment and mission lifetime are fairly standard.

RMF also succumbs somewhat to a "checklist" mentality and, like any other option out there, requires a complex series of judgment calls by engineers to weigh acceptable risks. These judgment calls can become bogged down in concerns between the system builder, the acquisition office, and the accrediting authority, all of whom may have different ideas of what's really necessary. They can get further bogged down as the project goes through the complicated DoD acquisition process and can become more of a paperwork exercise than a security exercise. For all these difficulties, however, it can still be a useful technique.

There have been many efforts to further refine and expand upon RMF. MITRE's Cyber Resilience Engineering Framework (CREF) works on the front-end problem of understanding how one should go about selecting security controls in the first place [3]. It lays out a series of cyber

Accesses	Effects	Techniques
Supply chain	Data loss/exfiltration	Securely parse and ingest data
Physical tamper	Data corruption	Component isolation
Malicious insider	Denial of service	Surface/component minimization
Credential theft		Data protection (at rest, in transit, in use)
Sandbox breakout		Authentication and secure control
Control hijack		Randomize, diversify, adapt
		Rapid replacement and reconfiguration

TABLE 1

MATRIX Brainstorming: Accesses, Effects, and Techniques.

resiliency goals, objectives, and techniques, and then works to map the controls to those techniques. The list of techniques is replicated in Table B.1 in Appendix B.

2.4 MATRIX STRUCTURED BRAINSTORMING

MIT Lincoln Laboratory (MIT LL) develops far more than just cybersecurity technology. Among other things, Lincoln designs and builds RADAR systems, bioagent detectors, optical sensors, advanced microelectronics, and sometimes the systems that those components go into as well. These designs and subsequent prototype systems may occasionally get fielded directly, and thus be subjected to cyber threats.

MIT LL is internally developing methods to properly manage cyber risk for these prototypes. Our goal is systems that are secure now, and and easily securable in the future as new defensive capabilities are developed and new offensive capabilities are discovered. The process, termed the MATRIX process, discusses when and how MIT LL security expertise can be usefully brought in at each stage of the underlying prototype's development.

We will not cover the entirety of the "Securable Mission Systems" MATRIX method here. We instead focus on the structured brainstorming it proposes as a way to explore the problem space. MATRIX uses a three-dimensional space with attacker access, attacker effect, and mitigation strategy as the axes. By pairing up one value from each axis, defenders can consider the likelihood of a given attacker access–effect pair and the efficacy of a given mitigation strategy to thwart it. Designers can explore the solution space and work to develop the best "set-covering" approach to mitigate the threats.

We review the attacker access, attacker effect, and mitigation strategy lists in the remainder of this section. They are summarized in Table 1.

2.4.1 Adversary Accesses

Here we list a handful of common cyber threats that one could imagine being applicable to any cyber system, space-related or otherwise. We do not attempt to reduce to any specific system here, nor do we discuss specifics of any techniques or weapons that an adversary could use. Pairing threats with architectural elements provides basic "threat vignettes" that we can then seek to mitigate.

The means by which an adversary can gain unintended or unauthorized access to the system can include the following:

Supply chain. Hardware, firmware, and/or software components of the system were compromised prior to delivery and integration, providing unintended functionality that an adversary can exploit. Supply chain compromises can trivially enable other access means (e.g., by providing an easy "breakout" method, or by adding unexpected, adversary-controlled credentialed access, etc.).

Physical tamper. An adversary is able to gain physical access to some component of the system and uses information gleaned from the system to achieve access elsewhere. This can be, for example, a means to conduct credential theft.

Malicious insider. A trusted insider with authorized access to the system acts adversarially. Malicious insiders are extremely dangerous, especially if they are unconcerned with capture and are able to carry out their effect completely before they are identified and stopped.

Credential theft. An adversary that successfully steals (or guesses) credentials can assume the identity and privileges of an intended user of the system. The adversary may not have the expertise of the user being emulated.

Sandbox breakout. An adversary is able, via technological means, to move through a boundary designed to isolate or limit the scope of a subcomponent's actions. The boundary the adversary moves through may be an externally facing boundary; this may be a means of initial compromise as well if an externally facing software service is vulnerable to attack, for example.

Control hijack. An intended control channel is taken over by the adversary. This access vector may be achievable in the analog or digital domains.

2.4.2 Adversary Effects

Adversaries generally don't break into systems with no further objective in mind. Broadly speaking, the effects an adversary could carry out, once access is obtained, are as follows.

Denial of Service. The adversary prevents the system from working. This can involve complete, permanent denial, or it can be more nuanced—partial degradation and/or temporary denial. In the non-cyber world, the former can often be achieved with well-placed explosives and the

latter by well-placed jammers. This can also include **subjugation** if the adversary is able to not only deny the owners access, but also gains access to the system and repurposes it for their own use.

Data Corruption. The adversary alters data that the system generates, stores, and/or transmits. The corruption's extent and goals vary widely based on the system. An adversary may simply wish to degrade trust in the system's integrity. In the case of sensor systems, the adversary may wish to inject false positives or false negatives into the data feed and/or the processed results.

Data Loss/Exfiltration. The adversary simply deletes data that the system is supposed to process and/or makes unauthorized copies of the data for their own use. Data loss can be viewed as a more obvious form of data corruption and/or as a more permanent form of denial of service.

An adversary may seek only to project an overt, credible threat, without actually carrying out one of these effects. If an adversary convinces the United States that a denial of service effect is feasible, for example, then the United States has effectively lost its ability to rely on the asset—even if the adversary never actually carries out the effect.

2.4.3 Mitigation Strategies

Broadly speaking, these are the mitigation strategies that should be considered when engineering a system. There are as many ways to enumerate the strategies as there are strategies themselves—indeed, we've already discussed both RMF and CREF—but we offer our own list for consideration as well. Our list of techniques seek to be very broad, general-purpose best practices that work to prevent unauthorized access and minimize damage should unauthorized access be achieved.

Securely Parse and Ingest Data. Mistakes will be made when code is written, and larger codebases have room for more mistakes. Mistakes on the "edge" of a program where it interacts with data from other programs are particularly harmful. Designers should minimize and simplify interfaces between components of a system. This leaves less room for a data consumer to make a mistake that could lead to vulnerability. This includes message-, network-, and application programming interface (API)-level interactions. Several of the following strategies build upon this basic concept.

Surface/Component Minimization. In the spirit of secure parsing and ingestion, the overall size and complexity of the system should be minimized. Remove unneeded features, libraries, modules, and other code that is not necessary for the system. Identify and aggressively minimize "trusted" components without which the system will fail to operate. These components should receive intense scrutiny during the design process to understand and limit the effects of their failure if compromised.

Component Isolation. Software generally has two types of interfaces: customer-facing interfaces through which it sends/receives commands and data to other software, and system-facing

interfaces through which it manages the accesses and resources needed to get its work done. The system-facing interfaces are often library APIs and operating system APIs. Those libraries and operating systems can be viewed as "sandboxes" designed to contain the software and ensure it doesn't disrupt other functions on the system. The principles of secure parsing, ingestion, and surface minimization apply when interacting with the system-facing world as well.

Data Protection (At Rest, In Transit, In Use). Designers should protect data as they move through mission systems. Protections typically focus on confidentiality, integrity, authenticity, "freshness" (including replay protection), and non-repudiation. These data include key material, credentials, the communications to/from the terminals and metadata about those transfers, such as the geographic locations of terminals. Some data that may not require protection for mission needs could nevertheless benefit security if protection is implemented.

Authentication and Secure Control. Developers should provide means to remove unauthorized data flows and ensure that command/control (C2) messages are only accepted from authorized entities. Authentication and secure control are important components of a system and of all of the other systems that interact with it. Ensure the system can accurately measure and report its internal state to appropriate parties for monitoring and remediation. This is a blend of surface minimization and data protection, used to "bind" data to specific entities and enforce relationships between them.

Randomize, Diversify, and Adapt. Where possible, create system diversity without increasing configuration and management burden. Diversity makes it more difficult for an adversary to achieve a "break once, break all" attack and requires the adversary to use more complicated attack vectors to overcome the moving-target nature of the system under attack. Ideally, engineer the system to automatically recognize which moving-target options are vulnerable to observed attacks, migrate to an immune option, and eschew the vulnerable options in the future. In the electromagnetic (EM) world, adaptive frequency hopping is the canonical example: the system automatically adapts to the EM spectrum by avoiding noisy channels, and it randomly jumps between clear channels to make it more difficult for the adversary to target the transmission.

Rapid Replacement and Reconfiguration. Ensure the system can be easily reconstituted in case of compromise, failure, or discovered vulnerability. When a vulnerability is discovered, ensure that the vulnerable component(s) can be rapidly and safely replaced. When an attack succeeds and mitigation is attempted, adjustments need to be made to ensure that the attack that took the system down won't succeed a second time, and it must then be brought back into service as rapidly as possible.

2.5 STAMP, STPA, AND STPA-SEC

It is tempting, but incorrect, to conflate reliability engineering with security engineering. Reliability engineering seeks to mitigate the impact of an arbitrary, capricious, uncaring universe. Reliability engineering considers metrics such as mean time to failure (MTTF) and mitigation

strategies such as redundancy. For example, redundant array of inexpensive disk (RAID) technology is commonly used to ensure that a single hard drive failure will not cause data loss.

Security engineering, however, concerns itself with malicious attack, where an active adversary effectively gets to choose a highly improbable stimulus. A system designed to provide only traditional reliability is unlikely to fare well when faced with adversary actions; RAID does not save you from deliberate, malicious data deletion, for example.

Designers may also conflate safety engineering with security engineering in the same way. These systems are generally more robust from a security perspective because they may consider mitigating external forces that are actively malicious, but they still lack a complete threat model.

However, safety engineering can still be an excellent place to start working on security engineering. Researchers at the MIT campus have been working on this very hypothesis and have yielded a trio of processes as a result: STAMP and STPA, which focus on safety, and STPA-Sec, which extends them to security.

2.5.1 STAMP

Professor Nancy Leveson is at the helm of MIT's System Safety Research Laboratory. She asserts that in the past, systems were sufficiently simple that the reliability and safety of individual components composed in a reasonably straightforward manner to inform the overall safety of the system as a whole. However, modern systems with their microprocessors and their corresponding unfathomable plethora of machine states do not lend themselves to this piecemeal analysis. Prof. Leveson's model, Systems-Theoretic Accident Model and Processes (STAMP), proposes that safety is increasingly a control problem, not a component problem.

In general, STAMP proposes decomposing a system into control-flow loops and then reasoning about the loops. The simple power of the approach is that it can be applied at almost any layer of abstraction and refined into arbitrary granularity as the exploration of the space and maturation of the system's design dictates. (We will put it to use in Section 4.2.3.)

Once the control structure is established, designers can examine potential deviations from the norm in a structured manner. STAMP encourages the consideration of four classes of unsafe control actions. The following description is excerpted directly from [13]:

- An unsafe control action is provided that creates a hazard (e.g., an air traffic controller issues an advisory that leads to loss of separation that would not otherwise have occurred).
- A required control action is not provided to avoid a hazard (e.g., the air traffic controller does not issue an advisory required to maintain safe separation).
- A potentially safe control action is provided too late, too early, or in the wrong order.
- A continuous safe control action is provided too long or is stopped too soon (e.g., the pilot executes a required ascent maneuver but continues it past the assigned flight level).

STAMP also makes several assertions about software that are controversial on the surface. First, software does not "fail" in the sense that a physical part can fail. Rather, software always does exactly what it says it does unless the hardware that it's running on fails. Second, that software's role in (non-malicious) accidents is almost always a result of bad requirements, not bad coding, so making the software better or more reliable will not necessarily improve safety.

For greater detail on STAMP, the brief overview [14] and the full-length book [15] are both available to interested readers for free. The book is also available in hardcover.

2.5.2 STPA

The STAMP work was further extended to Systems-Theoretic Process Analysis (STPA), [13]. STPA further explores component interaction accidents, where a set of normal or expected operations between components in a system can yield an unexpected or unsafe behavior. STPA takes the STAMP process and works to better determine which unsafe control actions may be problematic. The component interaction view takes into account the system-level phenomena that can give rise to accidents, above and beyond the reliability or failure modes of an individual component.

STPA starts by identifying hazards, which are combinations of a given system state and a worst-case set of environmental (external) conditions, that lead to an accident or loss. The key observation is that a hazard must include an aspect of the system that can be controlled as well as a set of conditions that may not be in the system's direct control. This helps to reframe each hazard as a requirement that the system must satisfy. A handful of worked examples are presented in [13].

2.5.3 STECA

STAMP and STPA are generally applicable to all levels of design. However, in the earliest stages of design, systems are often described predominantly in prose, and little is yet set in stone. The Systems-Theoretic Early Concept Analysis (STECA) technique is proposed in [16] to address this case as well. The paper provides guidance on how to systematically disassemble paragraphs of text describing a system design, convert them to pieces of the STAMP/STPA model (control flows, actuators, sensors, etc.), and then reason about them at that level of abstraction. The paper suggests that such an approach can derive more useful, finer-grained safety requirements than traditional approaches. To make its case, the paper uses an air traffic control system document as an example and supplies partially worked output of their approach vs. a more traditional preliminary hazard list and analysis approach.

2.5.4 STPA-SEC

Colonel William "Dollar" Young Jr. has worked to refine STAMP and STPA to address security concerns in an approach termed STPA-Sec [17, 18]. Among his foundational assertions regarding cybersecurity, he advocates for:

- Identification and control of undesired functionality, not just desired functionality;

- A focus on operational approach (strategy) rather than on engineering (tactics);
- A de-emphasis on protection ("shell" defense), instead favoring mitigation and elimination;
- Early, "left-of-design" assurance work, based on the flexible levels of granularity afforded by STAMP; and
- Mission assurance as the "gold standard of success," not cyber security.

This seems deceptively simple, and perhaps it is. STPA considers the threat posed by an uncaring environment and tragically errant humans. STPA-Sec uses the same analytical structure, but adds the threat of a deliberately malicious human. The basic analytic flow remains largely unchanged.

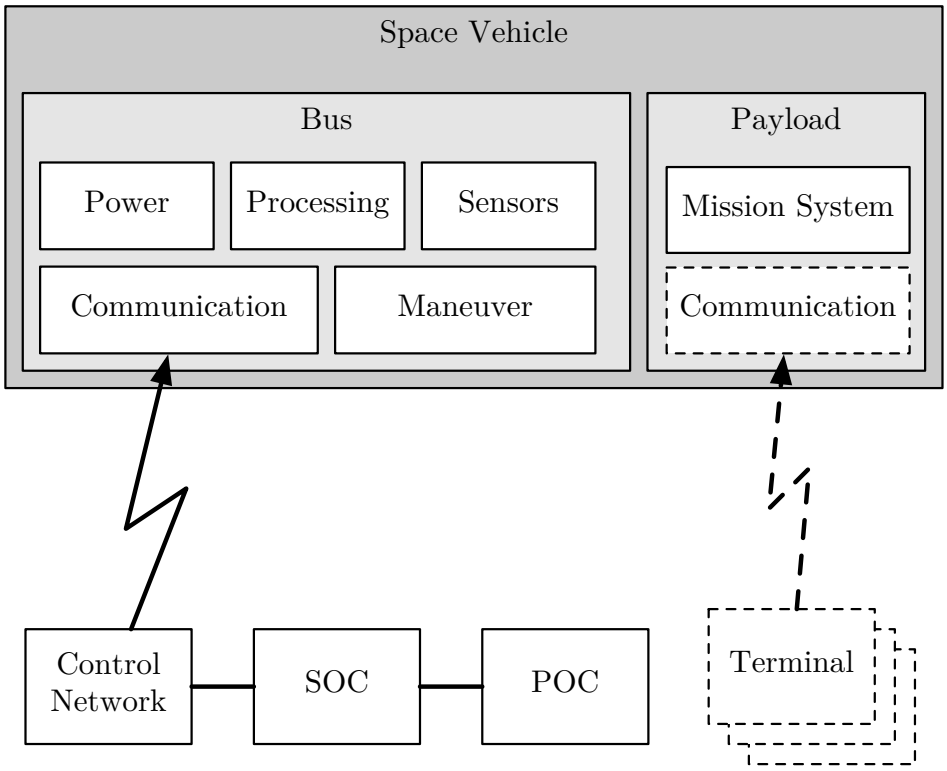

Figure 4. Diagram of typical SV and ground components, including the optional terminal and payload communications components.

3. OVERVIEW OF SATELLITE DESIGN

A satellite system comprises at least two, and sometimes three, basic architectural components: the space vehicles (SVs), which go to orbit; the ground control, and—for communications satellites, at least—the terminals capable of directly using the SV's capabilities. The ground control task is often split between a spacecraft operations center (SOC) that focuses on orbit main-tenance and maneuver and a payload operations center (POC) that focuses on payload control and management of payload-generated data. Different organizations may break those tasks down in different ways. Large and expensive ground components, like antennas, are often shared between missions via a control network. These components are summarized in Figure 4. The remainder of this section begins with the operational environment, then considers all of the components in turn.

3.1 SPACE ENVIRONMENT

The ground and payload control systems are often traditional enterprise computing environments, with traditional enterprise computing risks [19]. Terminals can exist in a wide variety of environments: forward-deployed, CONUS-based, mobile, fixed, manned, unmanned, on land, on

sea, undersea, and airborne—all environments more hospitable than the one the spacecraft must endure.

The space environment is a harsh place to park a machine [20]. Beyond the Earth, the magnetosphere offers no protection from the solar wind and background radiation. In high LEO orbits and above (anything higher than 1,000 km or so) satellites often must fly through, and survive, the Van Allen radiation belts. In lower LEO orbits, satellites must contend with atmospheric drag, damage from elemental oxygen, and charged particles that get past the magnetic field's protections.

The harsh environment takes its most serious toll on electronic components. Integrated circuits are particularly susceptible to radiation damage. There are two basic threats to circuits: total ionizing dose (TID), a gradual build-up of damage that eventually causes failure of the part, and single event effects (SEEs) caused by individual, high-energy particles that cause transient glitches and latch-ups in parts [21].

3.2 SPACE VEHICLE

The satellite, or SV, receives a great deal of scrutiny and engineering care, and for good reason. The SV must undergo the harsh rigors of depressurization and shock during launch, survive the temperature and radiation vagaries of space, and (with any luck) avoid strikes from space junk and other satellites long enough to accomplish its mission and survive for the duration of its design life.

A satellite is usually separated into two segments: the payload and the bus. The payload is the portion of the satellite present to carry out the overall mission—collecting sensor data, relaying communications, etc. The bus is the portion of the satellite dedicated to supplying the payload with what it needs to carry out the mission—proper orientation and orbit, electrical power, and communications.

The bus and the payload may be built by different organizations and may be owned and operated by different organizations as well. Sometimes a single bus carries multiple payloads—built, owned, and operated by multiple disparate entities—within the same physical satellite. This hosted payload model can be quite cost-effective for some users, but it brings with it a complicated set of additional security concerns. Hosted payloads are not addressed further here, though we return to the concept to motivate some of our single-payload recommendations.

3.2.1 Sensors

A satellite bus has some responsibility to sense its environment: without any ability to know where it is, where it's going, and what it's looking at, it risks being demoted to space junk in short order. Typically, the satellite cares about its handful of orbital parameters describing where it is and where it's going along its orbit (in Euclidean geometry, x, y, z, and deltas for each), along with the six angular measurements describing where it's looking and how it's rotating (pitch, yaw, roll, and the deltas for each). Satellite buses therefore routinely carry a smattering of sensors in order to maintain awareness of these parameters.

To sense its orientation, a satellite tends to use a set of coarse sun sensors to get an initial fix on the brightest object it's ever likely to see and at least one star tracker that surveys a small keyhole-shot of the heavens in an effort to identify known constellations. Position is often determined by GPS in concert with an inertial measurement unit (IMU). Position and orientation are critical knowledge for pointing communications antennas and solar arrays for optimal performance, among other things.

The speed of rotation is also important: too fast and one risks damaging spacecraft systems via excess centripetal force; too slow and one may be unable to carry out the mission. To sense rotational speed, the star tracker can be used once more, along with inertial measurement units. Difficulties can arise when multiple instruments measuring the same phenomenon disagree: inconsistencies between the IMU and the star tracker contributed materially to the loss of the Japanese Hitomi satellite [22] and may have also caused the problems observed with the NASA STEREO-B satellite [23].

Beyond knowing where it is and where it's pointing, a satellite must also understand its internal state. A satellite that is too hot or too cold risks damaging or destroying sensitive components. Satellites that can't sense overvoltage or undervoltage situations can't detect and react to component failures or latchups, and risk failures cascading throughout the system. Satellites therefore tend to fly a robust set of health and status sensors, and the data from these sensors are routinely transmitted to the ground as part of the satellite's telemetry feed. Particularly well-instrumented craft may even generate more data than the downlink can support, and fretful engineers on the ground must decide which sensors they wish to monitor and when.

3.2.2 Orientation and Maneuver

The vast majority of satellites require at least some ability to move in order to accomplish their missions. The simplest type of movement is *orientation*, or adjusting the satellite's facing. Almost all satellites possess this capability to some extent. The more complicated capability is maneuvering, or adjusting the satellite's orbit. Maneuvering can be optional for short-lived missions in LEO, but almost all other regimes require at least a modest maneuvering capability.

Satellites orient themselves in flight by stabilizing and controlling pitch, yaw, and roll as they travel along their orbits. If a satellite remains in a low orbit and only needs to stabilize in relation to the Earth, designers can use torque rods, or magnetorquers, to dump the satellite's angular momentum into the Earth using the Earth's magnetic field. Satellites that need to stabilize in all three axes or satellites that need to occasionally point themselves away from Earth will add a on each axis. The satellite can change its rate of rotation about an axis by counter-rotating the corresponding reaction wheel.

Orientation capabilities give satellites little to no opportunity to alter their orbits, however. In LEO, orientation can allow a satellite to "fly" somewhat through the extremely thin atmosphere, but there is very little control and "flight" of this sort can hasten the satellite's eventual reentry and fiery demise. Anything more sophisticated generally requires some amount of maneuvering capability. Satellites may need to maneuver for a variety of reasons, including:

- performing basic station-keeping to counteract drift and keep a satellite exactly positioned within a known and expected orbit;
- carrying out specific mission goals that may require orbit alteration—e.g., a satellite that propels itself from a geosynchronous transfer orbit (GTO) to GEO;
- dodging an imminent collision (e.g., with space junk on an intersecting orbit);
- intentionally deorbiting or boosting to a "graveyard orbit" when the spacecraft reaches end of life.

These maneuvers are carried out via steadfast obedience of Newton's Third Law: by forcibly ejecting some of the spacecraft's mass in one direction, the spacecraft can move in the other direction. There are a variety of technologies used for this purpose, from solid rockets to cold-gas thrusters, warm-gas thrusters, or ion engines [24]. All of these are generally measured in terms of delta-V, the total amount of velocity change that they can impose on the satellite. Designers also must weigh the maximum acceleration the propulsion system can impart as well as the mass required. Ion drives, for example, are extremely efficient, but can provide only a very small amount of force.

Sometimes a satellite must use propulsion for orientation as well. Reaction wheels can only "soak" so much angular momentum; they can only spin so fast. Satellites in LEO can use their torque rods to dump excess angular momentum. Satellites in higher orbits don't have this luxury and must use thrusters to bleed off the excess if something causes a gradual buildup.

3.2.3 Communication

It's a rare satellite that can accomplish its mission without ever communicating with anything else. As a result, satellite buses tend to carry at least one radio used for telemetry, tracking, and command (TT&C). Communications from Earth are "uplinks", to Earth are "downlinks", and from satellite to satellite are "crosslinks."

The data rates required for basic command/control (C2) are generally modest; some satellites get by with 1200 bps up and 9600 bps down, for example. Most communications is done via radio frequency (RF) links. The engineering tradespace includes not only the data rate requirements, but also the available size, weight[1], and power (SWaP) for the antennas and amplifiers, the encoding properties, the distance from the satellite to ground stations, and the "steerability" of the communications beam (by rotating the satellite itself, putting the antenna on a gimbal, or electronically steering an array of antennas).

3.2.4 Power

Satellites require electrical power for almost everything they do. RF amplifiers and receivers, propellant control valves, reaction wheels, processing units, and thermal management systems

[1] For spacecraft, this should be mass, not weight, but sadly, SMaP is not a standard acronym.

(survival heaters, active radiators) all require power. Satellites are on their own when it comes to power generation.

Solar panels are by far the simplest and most common approach to provide power to a satellite. The power system can be as simple as a set of panels on the surface of the spacecraft, or as complicated as deployed solar arrays on steerable arms that keep them optimally oriented for maximum collection. Onboard batteries are used to power the spacecraft while it is in eclipse, traveling "behind" the Earth and out of view of the Sun. The power budget for a CubeSat is generally very modest, and substantial effort is made to minimize power consumption (and thus simplify the power subsystem and reduce overall SWaP).

The rare SVs that don't use solar panels are usually interplanetary probes that may travel so far from the sun that solar panels can no longer generate sufficient power. For these satellites, a radioisotope thermoelectric generator (RTG) is used. An RTG uses a hot pile of plutonium and a thermocouple to generate electrical power. Plutonium is not cheap. As a catastrophic launch failure that scatters chunks of RTG far and wide would be pretty unpleasant, these systems are very carefully over-engineered, further adding to their cost and weight. They are therefore used only when nothing else will do. There are exceptions, of course, even some within Earth orbit [25].

3.2.5 Processing

As described in Section 3.1, the space environment does not make traditional computing easy.

Certain "rad-hard" processors have been carefully designed to resist large amounts of radiation damage (e.g., the BAE RAD750™ and its relatives [26]), and these processors are often the only choice for missions that require either a very long duration or a particularly inhospitable environment (GEO or beyond Earth orbit). These processors, while effective, are very expensive and underpowered/inefficient compared to commercially available parts.

Many SmallSats (and especially CubeSats) operate in LEO and have short mission durations. For these satellites, COTS processors are an increasingly viable choice. COTS parts can often resist fatal TID damage for several years and careful design of rad-hard "watchdog" circuits and other engineering choices can provide for efficient reset and recovery of a part that's been hit by a SEE. (See Section 3.1 for discussion of the threat of TID/SEE.) As a result, SmallSats have an almost unprecedented amount of onboard processing power that can be brought to bear.

With great power frequently comes great bloat—these COTS processors are often called upon to run COTS operating systems, bearing COTS libraries and software stacks. Variations of Linux, including real-time versions, are increasingly frequent choices. At least one amateur satellite flew using Windows CE .NET as its operating system (OS) of choice [27]. As a result, small satellites are increasingly subjected to many of the same issues as any other computer running a commodity OS.

3.2.6 Payload

The previous sections documented the various components of the bus, but buses do not drive for the bus's sake. The payload is the bus's (and thus the satellite's) *raison d'être*. Payloads have

their own demands on the satellite, chiefly SWaP requirements. Payloads may require the satellite to perform various physical maneuvers to ensure the payload can operate optimally—in some cases, the payload's desired orientation may not be sustainable and the satellite must routinely switch between an operational mode (charge-depleting) and a sun-facing mode (for recharge). The payloads may use bus communications capabilities to reach the ground or it may have its own, either because the payload requires higher bandwidth or because communication is the payload's actual mission.

3.2.7 Integration

Often a payload is "bolted on" to a suitable (but not custom) bus, and many vendors sell buses or bus-oriented "mix-and-match" components for these purposes. However, some payloads are tightly integrated with and around the buses that support them. Such an approach may add design complexity, but it often yields payoffs in terms of launch mass and size, enabling a more powerful payload to fly in a smaller form factor than it could otherwise inhabit.

The integration of bus components is itself a complicated task, spanning a variety of disciplines. Most relevant to cyber are the various hardware and data interfaces used to manage the various components of the satellite (see Section 3.2). Efforts have been made to standardize these interconnects [28,29], but industry is still largely in a "Wild West" period and has not settled down to these sorts of standards. In addition, the onboard processor may have its own surrounding support structures, such as hardware watchdogs, and associated board support packages to integrate into the control software.

3.3 GROUND CONTROL

There are several terrestrial roles involved in the care and feeding of a spacecraft:

- Command authority of the bus
- Telemetry processing from the bus
- Command authority of the payload
- Telemetry processing from the payload
- Orbit awareness (tracking) and maintenance
- Mission data processing (of data from the payload)
- Cryptographic operations for communications security (COMSEC) purposes
- Free-space communications

The first five tasks — collectively termed "Telemetry, Tracking, and Command" or TT&C —are typically computationally simple and can be handled by modest resources. Orbit awareness

can be challenging, depending on the satellite. Some designs may be able to use ranging signals to discern its location via triangulation, and others may need to rely on other sensors external to the system to identify location.

Mission data processing can be very demanding, depending on the payload, and may require large amounts of dedicated (or bursty) processing power on the ground to crunch raw data from the satellite into actionable information in a timely manner. Bursty processing needs lend themselves to a cloud-based processing model, with all of the traditional risks and rewards of such an approach.

COMSEC is a very common technique in space systems, so common that we document it here as well as in Section 4. Cryptographic operations are typically handled via dedicated, purpose-built hardware. From a design standpoint, it is often easiest to buy and use well-understood hardware and protocols for these tasks. Existing solutions ease accreditation paths, leverage extensive work on correctness of implementation, and reduce the risk of incompatibility between the ground and SV implementations.

Communication likewise requires special purpose hardware, usually RF amplifiers, modems, satellite dishes, and the like. The size of the dish required is a function of frequency, desired bandwidth, and the power of the satellite's own transmission and reception capabilities. Some of these dishes can be quite large—the UHF dish at NASA's Wallops Island facility is 18 meters across [30]—in order to minimize the on-orbit SWaP. (Naturally these trades differ substantially for satellites that have communications to less-advantaged ground stations as a mission objective; one cannot expect a warfighter to carry a handheld radio and use it with an 18-m dish.)

For especially high-speed communications, optical transmission is increasingly viable at distances far greater than LEO [31]. Optical may be an increasingly tractable option for smaller satellites. Because optical transmission requires precise pointing, it is likely that a low data rate RF link will still be maintained for TT&C purposes.

All of these tasks must be accomplished to properly fly and use a satellite, but it is not necessarily the case that all of these tasks are performed by a single organization. There are a few factors that motivate consolidation and outsourcing of some of these tasks: orbit deconfliction, outsourcing, and cost of RF equipment.

First, the organization that pays for a satellite typically cares most about the payload and the tasks that the payload is able to carry out. The POC therefore focuses heavily on care and use of the payload. They may not necessarily be interested in also having the expertise, staffing, and equipment on hand to fly and manage the bus.

As an additional task and responsibility, a satellite with the luxury of maneuvering will want to execute burns as needed to maintain their intended orbit. The ground support must maintain awareness of collision threats and execute evasive maneuvers if needed. This is a lot of work for an organization that is principally interested in the payload.

For these reasons, some mission owners outsource these tasks to aggregated ground stations like the Blossom Point Satellite Command and Tracking Facility. These SOCs "fly" multiple satellites, handling most of the bus-related tasks.

The POC issues commands through the SOC[2]. The SOC adds necessary commands for orbit maintenance and other bus-related tasks, performs any necessary COMSEC, and sends the commands to the satellite.

There is also a great benefit to aggregating the RF hardware necessary to actually communicate with a satellite. First, the hardware is often expensive. Second, for LEO missions, the dish can only contact a given satellite for a small portion of its orbit (when it passes overhead), meaning the antenna and gear would otherwise be idle for the remainder of the pass. Lastly, for LEO missions that need frequent contact, multiple antennas dispersed across the globe are necessary. All of this adds a lot of cost without a very high duty cycle. Organizations like the Air Force Satellite Control Network (AFSCN) thus serve as "antennas for hire," in a sense, providing a global network of antennas that can be leveraged as needed to command various satellites. AFSCN can also fulfill the SOC role for many missions.

In short, many of the ground systems used to execute TT&C tasks are geographically dispersed and shared resources.

3.4 TERMINALS AND CROSSLINKS

For satellites that communicate with entities other than their ground networks, the hardware and cryptographic concerns largely remain. We generically refer to these non-SOC/non-POC endpoints as "terminals."

"Terminal" generally evokes the image of an antenna on the roof of a HMMWV, a handheld antenna employed by a dismounted user, or an intrepid explorer wielding a large iridium satellite phone. In all cases, the terminal hardware is far more SWaP-constrained than the ground station's hardware, and the terminal is often at greater risk of compromise or capture. Communication with terminals is almost always a payload task, not a bus task.

Satellites may also communicate with other satellites, via crosslinks. A constellation of satellites cross-linked together might be able to get by with fewer ground stations, as a satellite not in view of a ground station can use crosslinks to reach a satellite that is in view. Heterogeneous satellites able to communicate with one another could enable even more elaborate (and convoluted) paths for data flows. Crosslinks may be bus- or payload-provided features.

[2] We use the terms in the Department of Defense (DoD) sense, but the U.S. Government is not always consistent with them: NASA, for example, uses SOC as "Science Operation Center" and attributes POC-like functionality to it, whereas the NASA "Mission Operation Center", or MOC, is much like the DoD SOC. Commercial providers may have completely different terms on a per-company basis. We apologize to readers accustomed to those or other definitions, but short of coining our own terms, we are stuck.

	Terrestrial	Space
Physical access	easy	very difficult
Environmental threats	power surges, sprinkler systems	radiation, thermal
SWaP concerns	low	very high
Installation method	IT professional	rocket launch
Communications	wired	intermittent wireless
User specialization	low	high
Platform specialization	low	high
Duration of unattended operation	days to months	months to years

TABLE 2

Differences Between Satellite Processing and Traditional Ground-Based Processing, Adapted from [2].

4. SECURE SMALLSAT DESIGN

There are a great number of ways to think about the process of engineering secure space solutions. Many start with approaches that apply to generic computing problems and work to tailor them to the space environment. The cybersecurity approaches one might prefer for a terrestrial system may be highly unsuitable in the space domain.

Design for spacecraft processing is especially challenging, and has a very different set of demands compared to traditional enterprise computing. Table 2 contains a summarized version of the differences between the two environments as documented in [2].

The ground segment, at first blush, is an ordinary enterprise network. The processing power necessary to manage satellite TT&C is often quite modest, but the infrastructure required to execute the communication itself—the dishes, modems, amplifiers, etc.—is not. As observed in Section 3.3, the communication resources may not always be co-located with the ground station, either.

The SV demands very different considerations. In general, satellites must place an extremely high premium on reliability and failover, as in-orbit maintenance is all but impossible. Spacecraft are not usually afforded the luxury of "detect and react", because of the difficulty in reacting to a compromised system: troubleshooting usually cannot include pushing a reset button or toggling the power switch. The spacecraft must take care of itself.

4.1 RELATED WORK

There are many existing works on security for space systems, especially the spacecraft themselves. We touch on a few of them here, and revisit as appropriate through the remainder of the document.

The RMF, as described in Section 2.3, is biased toward enterprise computing: there is, for example, an entire section on physical security that covers everything from visitor access policies to water shutoff valves. RMF provides for the concept of an "overlay" that tailors the RMF control set to a specific environment, so engineers working in that environment can begin their work with a set of controls and associated language that are better adapted to that environment. This standardizes the wheel somewhat and reduces the wheel-reinvention rate. The Committee on National Security Systems (CNSS) has released a "Space Overlay" for RMF use [2], and it includes a good description of the differences in design between enterprise and space systems in addition to its tailored control set (see Table 2).

Aerospace chaired a report that focuses largely on getting spacecraft acquisition through the necessary paperwork wickets (while also doing proper engineering, of course) [32]. The report's thorough writeup of process builds on some of the ideas from CREF [3]. The remainder of [32] details the space environment, offers a handful of recommended near- and long-term changes to effect greater security, and provides its own concept of the appropriate selection of RMF controls for the problem space. Although the path taken to get there is basically useful, the majority of the recommendations made are very high-level (e.g., mission level performance monitoring) or verge on platitudes (e.g., define and prioritize goals).

The Consultative Committee for Space Data Systems (CCSDS) has performed surveys of the threat [33] and made broad statements about architecture [34] and key management [35]. The architecture document focuses almost exclusively on cryptography at various layers of the network stack, but also has a section covering emergency C2 and safe mode considerations. The key management document does a good survey of the space, including sections for group-keying scenarios for disparate ground receivers, but doesn't necessarily go beyond what the RMF space overlay provides. The threat document motivates the cyber problem and proposes a coarse assessment of risks.

The Air Force Research Laboratory's Space Vehicles Directorate (AFRL/RV) authored a pair of "State of Space-Cyber" reports that identify a handful of high-level recommendations to improve security for space systems and drive those to more detailed cybersecurity technology development goals [36, 37]. The reports are excellent, but focus more on techniques than guidelines, and do not feature a companion instantiation at this point.

A report from Chatham House serves as a further call to action [38]. The report highlights the use of space systems by other critical infrastructure such as financial, weather forecasting, air travel, maritime traffic management, and communications. Their report proposes lightweight government regulation coupled with industry-led standards, asserting that this approach will yield the most rapid response. Although the document's carefully considered motivation is beneficial and the policy scaffolding it focuses on is a necessary consideration, it doesn't directly provide security solutions. We do, however, echo their desire for "future-proofing" designs to account for the typically longer development and operational timelines of satellite components and systems compared to most terrestrial technology.

4.2 EXPLORING THE PROBLEM SPACE WITH MATRIX AND STPA-SEC

We have surveyed a few techniques for targeted brainstorming of the cybersecurity space, and we have surveyed the general environment surrounding space-faring computer systems. In this section, we strive to bring them together as a way of developing the most interesting and underserved threat vignettes impinging on these systems and use them to motivate design guidelines that will improve the overall security of space systems built with them in mind.

4.2.1 Ground Rules

Exploration of the cybersecurity problem space is a dangerously overwhelming business. Numerous vendors, reports, and papers, including this one, offer their own maps that cover some portion of the wilderness. In an effort to further constrain the scope to something tractable, we propose the following assumptions and ground rules:

- The threat cannot break standard encryption algorithms (but the threat can try to steal keys, perform side-channel attacks. etc.).
- Specifics of a remote attack against a ground segment are irrelevant—at this level of abstraction, there is no difference between a successful spear-phishing attack vs. a client-side browser exploit vs. a misconfigured server, for example.
- The threat is not routinely mitigated by basic cybersecurity best practices that would be expected of normal enterprise operations or the appropriate mitigations for the space environment are substantially different from those chosen in more traditional enterprise environments.
- The threat is not impossible to mitigate by some amount of sensible defensive cybersecurity engineering. (This rules out things like kinetic strikes.)

In general, given standard best practices to encrypt C2 links, hacking directly into a satellite is not easy. The adversary's best option is to obtain access to the keying material for the COMSEC channel (either by getting the keys themselves or by gaining access to ground station equipment able to send data to/through the COMSEC device). The adversary may also be able to compromise the satellite's software load by persuading the ground station to upload corrupted software.

As a further reduction in scope, we deliberately neglect basic hardening techniques that are not unique to the domain. We will not talk to the screensaver passwords on ground systems, for example. We largely neglect hardening any one component of a satellite system for the sake of hardening. Instead, we focus on ways that the components can help to secure each other. Rather than working exclusively to secure a ground system from cyber threats, for example, one can also work to secure a ground system with the satellite's help.

4.2.2 Applying MATRIX

The MATRIX approach described in Section 2.4 has, as its core, a three-dimensional "space" of attacker accesses, effects, and techniques, summarized in Table 1. When the approach is applied

`prse`	Securely parse and ingest data	`supp`	Supply chain
`isol`	Component isolation	`tmpr`	Physical tamper
`smin`	Surface/component minimization	`mali`	Malicious insider
`prot`	Data protection (at rest, in transit, in use)	`cred`	Credential theft
`auth`	Authentication and secure control	`sbox`	Sandbox breakout
`rand`	Randomize, diversify, adapt	`hjck`	Control hijack
`redo`	Rapid replacement and reconfiguration		

TABLE 3

Abbreviations for MATRIX Elements Used in Table 4

at a more granular level — e.g., by specifying roughly which segment of the system is being attacked—it rapidly becomes challenging to consider all of the possibilities.

To manage the complexity, we neglect security solutions that are by and for the ground segment alone. Although there is certainly a large amount of useful work to be done there—more than enough for another report—this report focuses instead on the space vehicle (`sv`) and on interactions between the SV and the ground (`sv+g`) that can be mutually beneficial (or detrimental).

We further reduce complexity by reducing scope on the "effects" axis somewhat, lumping data corruption with denial of service (`deny`) and leaving data loss/exfiltration (`xfil`) as our two effect categories.

Given those restrictions in scope, we considered each of the $6 \times 7 \times 2 \times 2 = 168$ possible cases, as shown in Table 4. The handful of cases identified with solid boxes were deemed interesting for further exploration. The remainder of this section touches on each in turn.

We emphasize again that *cells in Table 4 that are not highlighted are nevertheless important for system security.* This document seeks to identify and explore promising areas that may benefit from nonstandard, domain-relevant mitigations. The vast majority of unmarked cells are still important to overall system security, but may be largely solvable using established techniques and/or using new techniques that require minimal tailoring to the system. We cover many of the unmarked boxes in a few broad strokes and then dive more deeply into the marked boxes.

There is little highlighting in the `sv+g` domain. To a large extent, issues with ground station security are not aided by spacecraft participation—for example, there's not much a spacecraft can do to help a ground system quickly reconstitute itself. There are a few exceptions where the satellite may be able to aid the ground station in its security, and we highlight them below.

Traditionally, COMSEC is used between the ground system and the spacecraft, and its practice essentially removes control hijack from consideration at that level. Hijack within the ground system itself is still a threat, but there is there is little unique about that threat here.

		supp		tmpr		mali		cred		sbox		hjck	
		deny	xfil	deny	xfil	deny	xfil	deny	xfil	deny	xfil	deny	xfil
prse	sv	■											
	sv+g												
isol	sv												
	sv+g												
smin	sv												
	sv+g												
prot	sv	■											
	sv+g							■	■				
auth	sv												
	sv+g							■	■				
rand	sv	■											
	sv+g												
redo	sv	■								■	■		
	sv+g												

TABLE 4

Brainstorming Figure for MATRIX Approach, Using Abbreviations from Table 3

*Highlighted cells are considered interesting based on the criteria in Section 4.2.1.

Hijack within the SV could be a possibility, but the mitigation strategies (e.g., star topologies for C2) are not unusual.

We will revisit these and other areas of overall spacecraft security in the Appendix, where we describe the instantiation we seek to build and describe techniques chosen to honor many of the defensive rows in Table 4 and the techniques in Table B.1.

The marked cells, however, offer opportunities for defense that are more specialized to the domain. We next summarize a few scenarios that cover the marked cells and propose solutions that could reduce their impact. We assume that perfect protection is not possible and focus on cases where an adversary has already achieved *some* level of undesirable access and we are working to mitigate the impact. The focal points remain the SV and areas where the SV and ground can work together to strengthen the system as a whole.

Supply Chain vs. Spacecraft. This scenario covers an adversary that has made an unauthorized modification to the satellite. This could be pre-launch hardware modification, pre-launch software modification, or post-launch software modification (a malicious software update). The adversary intends to use the supply chain compromise to trigger a denial of service at a later date by temporarily or permanently disrupting the spacecraft's function.

Unlike terrestrial systems, a successful hardware supply chain attack is effectively impossible to mitigate—the damage remains for the entirety of the SV's lifetime. Additionally, certain types of software may be unalterable after launch, or there may be types of malicious post-launch activity that can subvert the onboard software in an unrecoverable manner. In general, the fact that the defender can't physically access the system to recover it to a known state can motivate different engineering choices.

Malicious Insider/Credential Theft vs. Ground System. This scenario assumes an adversary is either a malicious insider or has obtained equivalent levels of access and information via other attacks. The adversary wants to "pivot" their access and gain a foothold on the SV.

There is ample work on insider threat mitigation for terrestrial systems of all types. The complication is the interaction between the ground system and the SV. Any malicious action that the insider can take on the spacecraft may be challenging to mitigate. These actions could include dangerous or unintended commands or the actual execution of a supply chain attack (uploading unauthorized software or firmware).

Sandbox Breakout vs. Spacecraft. This scenario assumes an adversary has obtained access to some part of the spacecraft. That access is not sufficient to accomplish the adversary's ultimate objective, but it is sufficient to permit pivoting attacks to take place that seek to gain further access on board. In this environment, this may be an adversary that has compromised a payload and is attempting to take control of the bus as well, as an example.

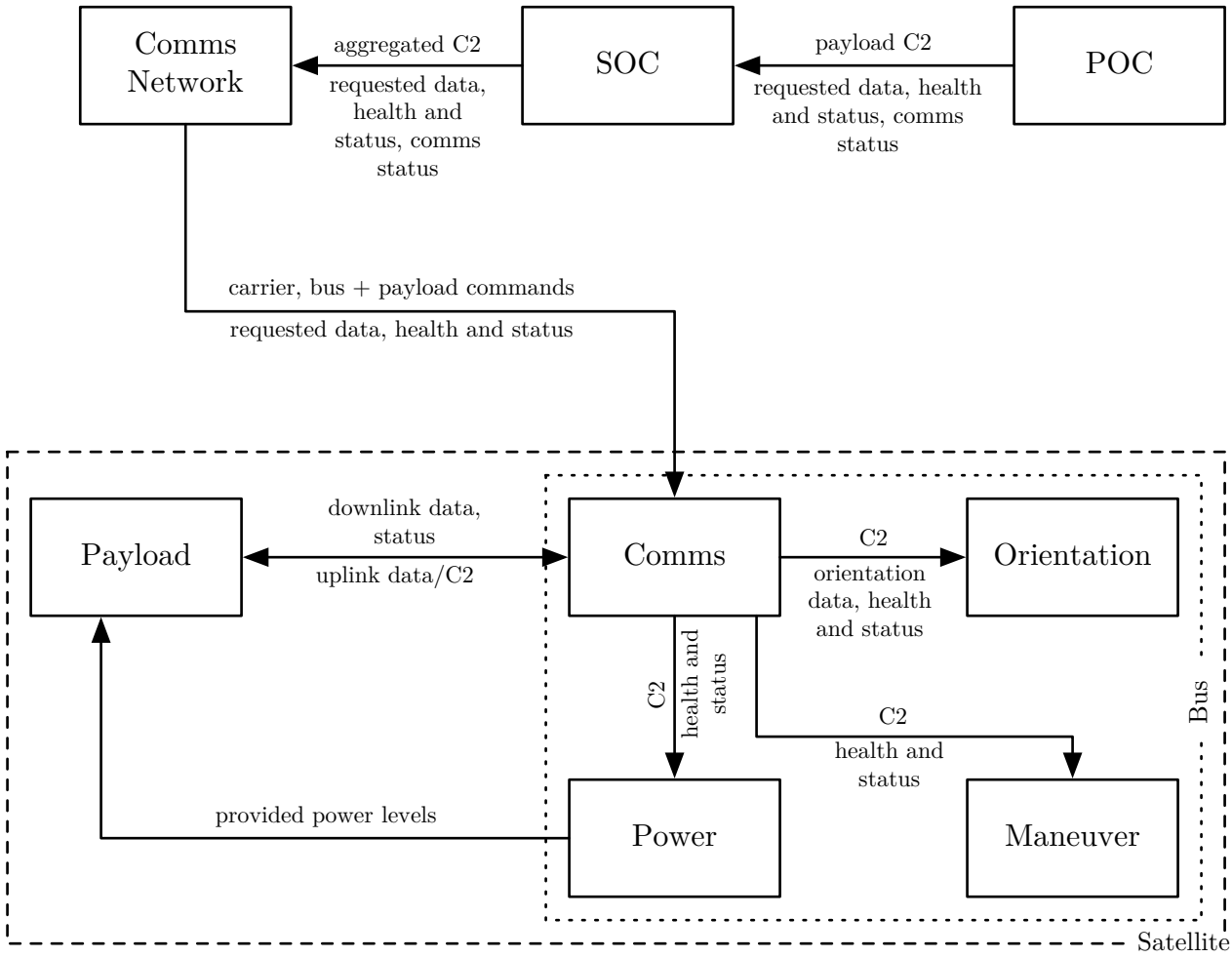

Figure 5. STAMP model of generic control loops—first cut.

Again, there is plenty of existing work on adversary footholds, and many of the common techniques (surface minimization, etc.) are therefore not marked in the table. However, the process for restarting a satellite to recover control can be a complicated matter. In particular, even if a satellite can evict the undesired access by rebooting the system, the sheer amount of time required to execute the reboot may itself jeopardize the mission.

4.2.3 Applying STPA-SEC

The STPA-Sec process lends itself very well to the level of abstraction with which we are working. We begin by breaking the system down into its five basic pieces described in Section 3, and then add further detail until we reach a point of diminishing returns.

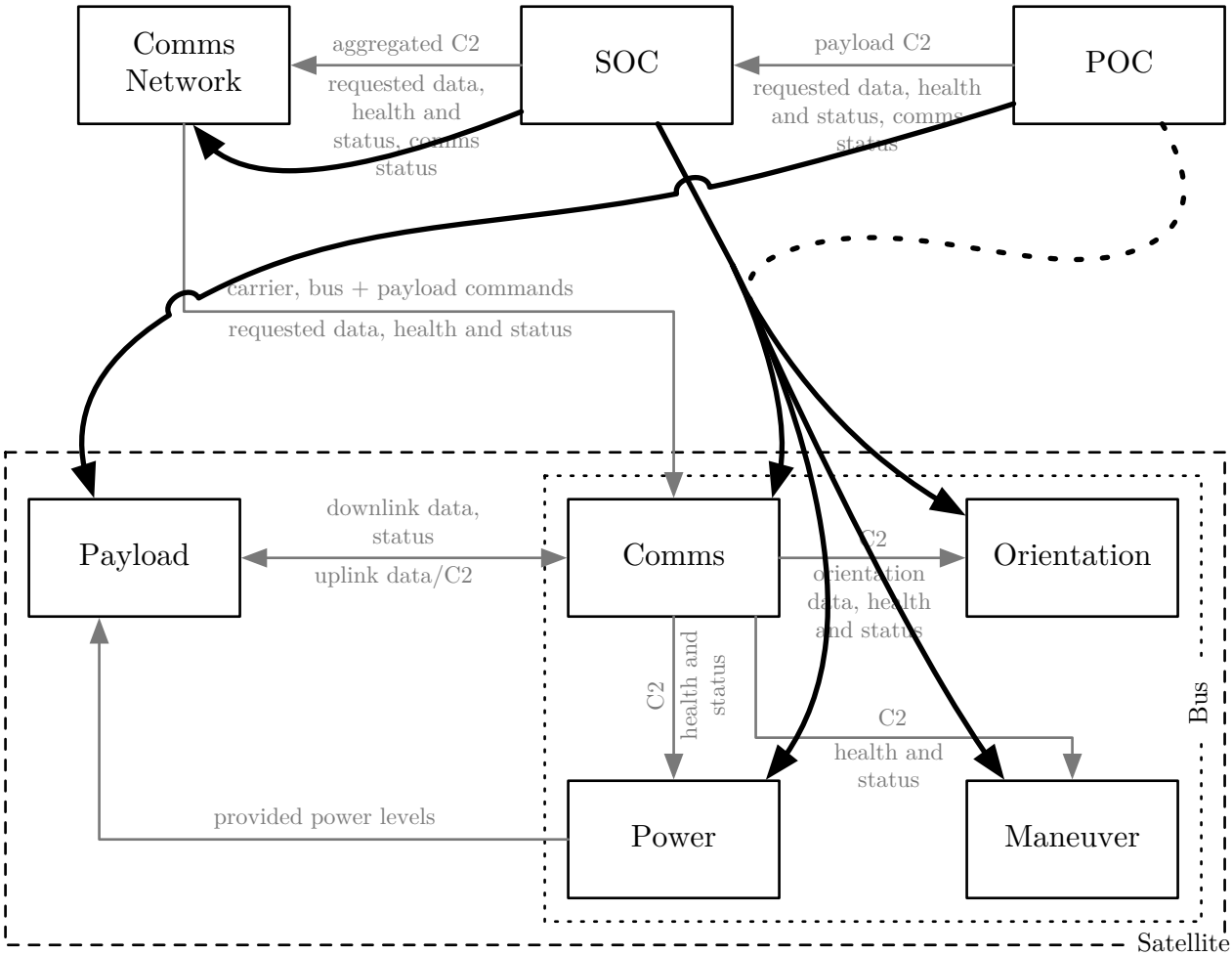

Figure 6. STAMP model of generic control loops—second cut.

The first cut at a control diagram for an SV is shown in Figure 5. Where "health and status" is shown, the implication is that the given spacecraft node is reporting on its own health and status in full detail—for example, the "maneuver" subsystem will report on reaction wheel speeds, propellant levels, valve positions, exhaust bell temperatures, etc. Where "C2" is shown on the spacecraft, it is intended to represent C2 for that specific recipient only. On the ground side, C2 is an aggregated quantity, insofar as the comms network gets all of the C2 destined for both itself (which dish to use, which frequency, etc.) and for the satellite it's speaking to.

There are a number of things to dislike about this first figure. Some of them are by design: for example, the common COMSEC solutions used to protect traffic between the comms network on the ground and spacecraft comms are not shown. This is by design. For STAMP-style evaluation, we will posit the impact of incorrect control flow over the link from the ground to the spacecraft, generate requirements about that control flow, and then see how COMSEC may satisfy those requirements.

The difference between C2 *paths* and C2 *authorities* is also key. Figure 6 attempts to overlay the control authorities picture. The clear disconnect between the flows in Figure 5 and Figure 6 informs the needed data protections to ensure that entities that should not command the spacecraft are denied that ability.

STAMP analyses do not provide a hard limit to just how far one can peer into a system design —the diagrams can get arbitrarily complicated and tend to grow in size and scope as a system's design matures. At this level of abstraction, we can develop the broad-brush guidelines and insights that we seek. Implementers should continue to expand and refine their analyses as they dive more deeply into their builds. Even if the security benefits were nonexistent, STAMP is considered a useful safety tool and should be seriously considered on that merit alone.

An even broader control authorities picture could explicitly take into account the owners of the spacecraft (bus), the payload, and the mission—i.e., the ultimate user of the payload's functionality. We don't show them explicitly or "wire them in" to the picture because the interplay between those offices is an organizational challenge solved different ways by different people. The one nod we make to this complication is in the dotted command path in Figure 6, acknowledging that a POC often has some direct command authority of the bus as well as the payload, e.g., if the payload and bus are owned by the same entity.

This page intentionally left blank.

5. SELECTED THREAT SCENARIOS

There are many threats one can imagine against cyber systems. In theory, the MATRIX brainstorming method produces $5 \times 6 = 30$ hypotheses to consider, and those are coarse-grained. Table C-1 in [32] lays out more than 50 combinations of attack vector, direction, medium, and method. STPA-Sec can produce anywhere from a tiny handful at a high level of abstraction to an overwhelming quantity as one dives more deeply into a specific system.

The MATRIX brainstorming from Section 4 highlighted three basic threats: supply chain attacks against the spacecraft, insider/credential attacks against the ground station, and "sandbox breakout" from a malicious beachhead on board the spacecraft. In this section, we further refine them into scenarios that we feel are potentially survivable or mitigatable.

The observations made via STPA-Sec regarding command structures motivate the use of cryptography to enforce authorities despite the typical differences between C2 hierarchy and communication hierarchy. STPA-Sec also helps users identify critical portions of the system, the ways the adversary could reach them, and the ways an adversary could spread—touching on the "Sandbox Breakout vs. Spacecraft" MATRIX scenario. We do not explicitly address the STPA-Sec findings or the sandbox breakout scenario here, but consider them throughout the development of the guidelines, as they color the pre- and post-attack environment for all of the other scenarios we do discuss below.

5.1 SCENARIO: TRANSIENT LOSS OF CONTROL OF A GROUND STATION

This corresponds to an insider threat vs. the ground station. We further restrict the threat to a transient loss, i.e., we assume that the threat is eventually identified and removed and that ground control systems can be restored to proper state. If we cannot regain control of the ground station promptly, then there are larger problems to contend with, and there's little point to engineering a secure spacecraft.

If C2 to a satellite is properly encrypted and authenticated, then there are a limited number of ways that an adversary could expect to inject legitimately formatted, yet maliciously intended, commands into the satellite itself. The adversary must either obtain some access to the COMSEC keys or obtain access to the ground network authorized to format and transmit commands through the legitimate COMSEC to the satellite.

If the adversary succeeds, then they can launch properly formatted commands to the satellite to carry out their own desired ends. This runs the full gamut of potential effects, from temporary disruption of service to subversion of the system or destruction (by issuing commands that permanently disable the satellite).

The keys to surviving this scenario are:

- minimizing the attack surface with respect to malicious commands,
- identifying that malicious commands are being sent,

- promptly evicting the adversary and recovering C2 authority, and
- surviving the malicious commands long enough for on the ground to recover C2 of the satellite and issue proper commands to restore safe operation.

5.2 SCENARIO: PRE-LAUNCH SUPPLY CHAIN ATTACK AGAINST THE SATELLITE

An adversary able to manipulate the software or hardware in the satellite before it reaches orbit could potentially install a backdoor that they could later take advantage of from the ground. There are many ways this could happen. Hardware- or software-based supply chain attacks against the satellite's components or the tools used to design and implement those components (compilers, synthesis tools, etc.) are perhaps the most obvious. Substitution attacks during the time between satellite completion and satellite launch are also a possibility, depending on the path the satellite takes from assembly to orbit.

This is a portion of the supply chain threat from MATRIX, but we've broken out the pre- and post-launch scenarios due to rather substantial differences between them. The post-launch scenario is covered in Section 5.3.

This scenario is challenging to survive if the adversary succeeds. The keys to surviving this admittedly difficult scenario are:

- minimizing critical components,
- focusing supply chain risk management efforts on those components,
- engineering for and rehearsing the on-orbit identification of non-critical components that do behave badly, and
- engineering for and rehearsing the on-orbit remediation of those misbehaving components.

The guidelines we propose and solutions we implement to address this concern will naturally need to take into account the control flows identified in the STPA-Sec brainstorming process and shown in Figures 5 and 6.

5.3 SCENARIO: POST-LAUNCH SUPPLY CHAIN ATTACK AGAINST THE SATELLITE

Once the satellite is in orbit, software updates may still be delivered. (Indeed, this is common practice for satellites that take a long time to get from launch to their operational location.) These software updates could be subject to many of the same attacks that were plausible pre-launch.

The keys to surviving this scenario are similar to those for a pre-launch attack. Software certification and authenticated flow from vendor to spacecraft reduce the risk, but do not eliminate

it. If the satellite is engineered to detect misbehaving (non-critical) software and is able to regain control and reimage the software upon direction from the ground, then this threat can be largely mitigated.

This page intentionally left blank.

6. DESIGN GUIDELINES

We leverage the brainstorming done on the scenarios in Section 5 to inform our proposed design guidelines. In all cases, we largely omit basic security practices (screen savers, password change requirements, badges for ground station personnel, etc.) and instead focus on design aspects that are either unique to satellite environments or unusually critical (for security) and rare (in implementation).

The result is eight design guidelines that can inform both satellite engineering and future research and development of defensive techniques specifically tailored for and responsive to the needs of satellite systems.

6.1 FAIL SLOWLY

Not all satellite commands are created equal. Some commands are fairly benign, from a satellite survivability standpoint, and some are particularly damaging. From the very beginning, satellite designers should work to minimize the set of damaging commands, and (where they are unavoidable) work to maximize the amount of time between when the command is issued and when the damage is unavoidable.

There are two types of damage to consider: mission damage and spacecraft damage. It is possible to damage a mission without actually harming the satellite itself—for example, commanding the satellite to turn its payload off will certainly disrupt the mission, but it is unlikely to have any impact on the satellite's longevity or ability to carry out its mission in the future. Designers need to have both mission needs and satellite needs in mind; we focus on the needs of the satellite, as they are more generic in nature.

Unfortunately, there are a great many ways to harm a satellite, largely as a side effect of their location in the void of space. Designers should consider the following, for example.

- Choosing propulsion systems that have a maximum propellant flow rate as small as possible, which correspondingly increases the amount of time it takes for the satellite to deplete its propellant.
- Choosing a C2 antenna with as wide a receive beam as practical so it's harder for the satellite to point away from the ground station.
- Choosing reaction wheel rates as small as possible, which correspondingly increases the amount of time it would take the satellite to point its C2 antenna in a bad direction.
- Including safety interlocks—in hardware where possible, and in software otherwise—to prevent combinations of unintended operation. For example, a satellite should not be capable of firing multiple opposed thrusters at once, nor should it be able to perform constant high-frequency changes to a reaction wheel's RPM (needlessly reducing its service lifetime).

- Queuing hazardous commands for as long as possible before execution, ensuring the queued commands are clearly reported in telemetry, and providing the C2 capability to rescind queued commands.

Many of the above suggestions are decidedly non-cyber design decisions. Solid engineering will generally minimize these variables anyway, but the additional reinforcement is beneficial.

It is impossible to eliminate dangerous commands. Some satellites may by their very design have "disposal" command sequences that may be irreversible—for example, once a satellite executes a deorbit burn, there is little to do but watch for the streak in the sky as it's incinerated. However, engineering to minimize these commands and minimize the immediacy of their effects will make the remainder of the security engineering easier to do—at a minimum, it reduces the urgency of any detect-and-react options that defenders may be able to employ.

6.2 GO BEYOND COMSEC

Communications Security (COMSEC) is the first step most space systems take for security's sake. An encryptor is placed at the ground station and on board the satellite, and all traffic between the two points is encrypted—essentially creating a VPN between the ground system and the satellite. Depending on operational need, transmission security (TRANSEC) and anti-jamming strategies may also be used.

COMSEC is a good idea, so it's no wonder that it's a commonly required feature and that there are multiple solutions available to implement it, including Innoflight's Space HAIPE product and Raytheon's KI-55 Gryphon. There are also efforts to standardize space-specific interfaces [39] and cryptographic formats at other layers [40] as well, but uptake is unclear. COMSEC provides a well-understood mechanism to largely eliminate the threat of a rogue ground station establishing a connection with the satellite or eavesdropping on legitimate communications.

However, COMSEC is generally very coarse-grained, providing authentication only to the level of granularity of the endpoints. In the simplest configuration, anything capable of talking to the red side of the COMSEC device can command the satellite. It's possible to construct a properly isolated ground system in that environment, but due to the increasing push for internet-functionality, isolated systems rarely stay as isolated as the initial design would indicate.

The coarse-grained environment that now exists is effectively equivalent to giving every entity on the ground station root/superuser access on the satellite. It is certainly possible to be more nuanced than this. We propose the common security paradigm of role-based access control (RBAC) to improve the granularity of control on the ground by specifically declaring which entities are permitted to issue which commands, and when they are permitted to do so.

Importantly, we need the RBAC policy to be enforced either on the spacecraft itself, behind the COMSEC, or on the ground as close to the COMSEC device as possible. Placing the RBAC on the ground means that some ground entity must be wholly trusted—with essentially no modification to the satellite, this is merely a restructuring of the ground station that puts all its trust in both the COMSEC and in the RBAC arbiter. Placing the RBAC on the spacecraft means that the ground

systems' trust can be distributed to the various people, credentials, and devices with which those people use those credentials, but it also means that the satellite itself becomes more complicated in design and construction.

To support spacecraft-based enforcement, we advocate for crypto beyond COMSEC, namely, cryptographically enforced RBAC. In this model, the entities possess cryptographic tokens that identify them and bind them to the roles they are permitted to exercise—and there are entities with the privileges necessary to change the access control policy itself.

Different systems may choose to do this in different places based on their own risk aversion and requirements. Systems that could be commanded from only a small handful of highly secure locations may choose to place the enforcement on the ground. Systems that could be commanded from a variety of locations may choose to place the enforcement on the satellite. In the extreme case, it may be possible for some systems to dispense with traditional COMSEC altogether and opt to directly leverage the cryptography mechanisms and associated key management to provide secure communication to and from the spacecraft.

RBAC implemented in this way will permit a wide variety of security options:

- Manifold entities can be authorized to safely perform non-hazardous commands, potentially more than would otherwise be possible in a COMSEC-only scenario.
- Hazardous but routine commands could require two or more people to use their associated tokens and collaboratively issue the command. This could cryptographically implement two-person integrity.
- The portions of a ground station that must actually be able to issue commands can be more easily isolated from other portions of the network, reducing the sheer size of equipment that must be air-gapped.
- Satellite owners could hand out sets of keys to ground controllers: a routine set, used in day-to-day operations, and an emergency set, stored offline in a physical safe (thus reducing the likelihood of compromise). The routine set could permit, e.g., consumption of a small fixed quantity of propellant per 24 hours, but in case of emergency, the ground station could quickly retrieve and employ the emergency credentials to expend additional propellant. The satellite would log the emergency use and report it in telemetry[3].

One example of this approach is shown in Figure 7. This example shows separate credentials (shown as keys) for several different roles. The ground system is broken into "Restricted Bus C2" for routine commanding, "Full Bus C2" for extraordinary commanding and non-routine tasks, and "Payload C2" for separate command and control of the payload resources. It also shows a "root of recovery" (RoR) key stored offline. These keys could be used for a variety of purposes discussed in Sections 6.3 and 6.5.

[3] This is usually termed "break the glass" RBAC, in which someone is permitted to exceed their normal rights in an emergency, with the knowledge that they'll be held accountable for their actions. Secure auditing is essential for these schemes.

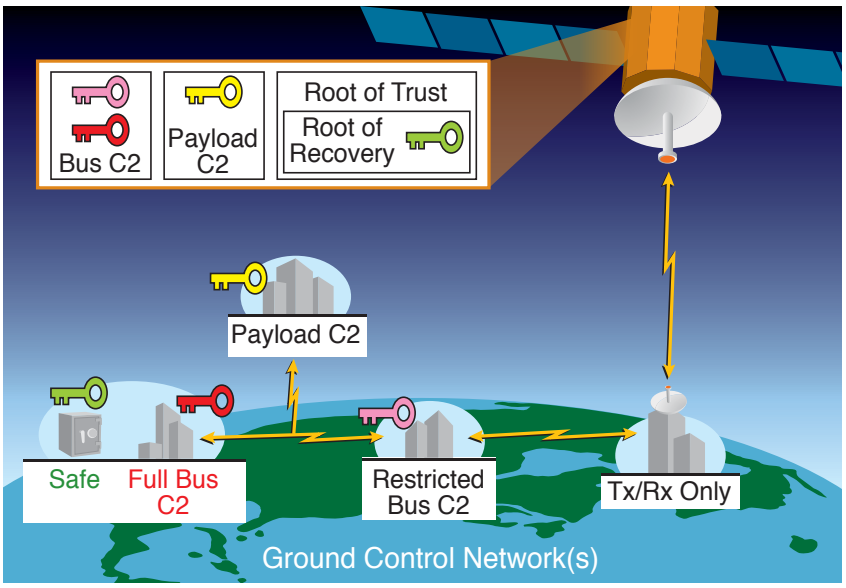

Figure 7. Notional key distribution for crypto-enforced RBAC approach.

We argue that the simple "if the COMSEC key is good you have full control" access control model is too coarse-grained. However, designers shouldn't overcorrect: when it comes to access control, it's entirely possible to make too much of a good thing. An RBAC scheme that accurately and precisely implements a least-privilege design may be too inflexible to accommodate slight variations in system needs, and even if it is sufficiently limber in that regard, a complicated design is harder to get right both in design and in implementation.

6.3 ABLATIVE DEFENSES

On a terrestrial system, if the complicated RBAC-based security solution proposed in Section 6.2 goes awry, operators could potentially disable some of the security system, accepting increased risk, to avert mission failure. This typically presumes some level of "superuser" access, which may be achieved with credentials but is more often achieved with physical access.

On a satellite system, a flaw in a complicated security solution runs the risk of loss of control of the satellite in its entirety. The likelihood of this result can be minimized by careful engineering, of course, but the consequences are so high that designers are naturally reluctant. It is very common for satellite designers to balk at components that have established no space heritage, an unfortunate Catch-22 that prevents components from flying in space unless they've already flown in space.

Bias against implementation risk is very sensible for spacecraft design, so any security solutions would do well to respect that bias and work within it. To that end, it may be prudent to include a way to turn the complicated RBAC system off and fall back to a COMSEC-only approach. Although this is arguably less secure, it may be preferable to the total loss of the satellite.

We therefore propose that all security systems onboard—not necessarily just the crypto—be designed in ablative layers, such that a ground controller could order the outermost (most complicated and effective) layer to be jettisoned for the sake of operational need, leaving the remaining layers intact. This dovetails nicely with the general principle of defense-in-depth design.

Another way of looking at this is as a deliberately engineered backdoor that can disable some security systems. This is an unattractive view, of course, but it may be an acceptable design decision when weighing the risk of having the security vs. the risk of not having the security.

As a method for implementation, we propose a set of jettison codes that can be transmitted directly to the satellite. These codes would be very tightly secured on the ground. If the decision were made to employ one, it could be transmitted to the satellite to deactivate the given layer. Designers must weigh the merit of transmitting this via the COMSEC channel (more complex) vs. sending it in the clear (less complex, but irreversible due to the increased replay attack risk).

Ideally, as security solutions for spacecraft mature and earn increasing hours of space heritage, the need for ablative design may itself abate.

6.4 FIELD AND LEVERAGE TELEMETRY

Satellites routinely collect and transmit extensive health and status measurements. These measurements are typically taken for environmental resiliency purposes, so ground controllers can preemptively alter operations to avoid hazardous conditions as they develop but before they can cause damage.

From a security standpoint, it is attractive to have measurements taken by something other than the measured volume. For example, one would not trust a list of processes from the operating system itself, as malware can and does easily doctor the list to hide its existence. Satellite designers should take advantage of the many sensors and processors on board to implement cross-measurement approaches.

To further secure the measurements, a satellite design will ideally use cryptographic protections at the point that the measurement was taken to prove to the ground station that the measurement was not elided, duplicated, or altered along the way from the measuring subsystem through the satellite's main processor and radio to the ground.

This practice can more generally be thought of as secure audit of aspects of the satellite's state, both of telemetry and of commands issued to the spacecraft and queued for execution. Ideally, every command should be automatically acknowledged/recorded in a write-only manner. Keeping and transmitting a full audit log could be impractical, so opportunities for clever engineering await.

6.5 INCLUDE A ROOT OF RECOVERY

Table 2 highlights many ways that spacecraft processing differs from traditional ground-based systems. Chief among them is the fact that the spacecraft is all alone. For a terrestrial system,

the ultimate root of trust—and root of recovery—is a human being with a screwdriver. The human can always unplug, reimage, replace, or otherwise take over a computer, given physical access. With a satellite, the human cannot be the root of recovery because the human can only exert control via the satellite itself.

The spacecraft must be its own root of recovery.

A root of recovery is not the same thing as a root of trust. A proper root of trust can do its job by ensuring that a system only starts up in a known approved state, and if that is not possible, the system doesn't start at all. If the system fails to start, a human (the root of recovery) is expected to reimage, repair, or replace the computer as appropriate. Spacecraft can't be treated this way.

From an engineering standpoint, this requires identifying a minimal subset of the satellite that can be trusted to faithfully carry out its mission, no matter what. The SV must be engineered such that its ground control station can remotely access, actuate, and rely upon the SV's built-in root of recovery to:

- know that the satellite is behaving incorrectly, either by its own means or by communications from the ground;
- assert control over a misbehaving element by resetting and/or disabling the element; and
- where appropriate, provide a misbehaving element with a different software/firmware pack-age (from the ground or from a previous onboard copy).

As documented in Section 3.2.7, there are critical touchpoints between a bus and the payload that it supports. We suggest that the bus should also be the root of recovery (RoR) for the payload, and indeed with a control flow breakdown like that in Figure 5, this is plausibly enforceable. We propose that bus designers should design their buses with the assumption that the payload may be hostile. This is a prudent choice in a hosted payload scenario, where a bus is called upon to manage multiple disparate payloads at once, each with different ownership, C2, and provenance. However, it is also a prudent choice in a single-payload system. The bus must be able to survive any attack staged from the payload and (if necessary) sever power from the payload to stop the assault.

A payload with no power is probably not very good at accomplishing its mission. The bus must be the RoR for the payload. Not only must the bus be able to cut power, but upon restoration of power, the payload must come up "clean" and accept software and configuration updates from the bus before resuming operations. These updates could be cryptographically signed by trusted sites on the ground before upload to the bus for use, for example. The payload could contain a "ratchet" value that prevents inadvertent reuse of old or outdated updates.

Beyond just a payload compromise, components of the bus itself may suffer from malicious faults. Engineers should aim to make the RoR as small as possible, certainly smaller than the entire bus. To a certain degree engineers already think this way: they routinely build in a safe mode for

satellites. In the safe state, the satellite shuts down all non-essential components, orients itself for maximum solar power, and awaits further instructions from the ground. This is an excellent design principle to ensure survivability in the face of faults or non-malicious issues, and it is an excellent starting point when designing for attack survival within the bus.

A bus safe mode could be engineered to presume malicious attack rather than merely presuming that bad luck has caused a part to fail prematurely. Thinking adversarially will motivate different decisions regarding the redoubt executing the survival-critical functions and how they are connected to other satellite systems. The designer must engineer ways to reimage software on components that may not want to be reimaged. All non-volatile storage must be accounted for, and a procedure for scrubbing it must be identified and employed—any data that could persist across a reset are potential mechanisms for continued mayhem.

Engineers should consider a potential recovery mode level that may demand even fewer functional parts than safe mode. Perhaps the recovery mode does not enable any orientation capabilities —no star tracker, no reaction wheels—relying only on the magnetorquers to orient roughly to-ward Earth—and does not power up any communications except for the barest minimum low-gain antenna and necessary crypto. The recovery mode might use more power than is generated and thus is not be survivable long-term, and the satellite may be forced to try for safe mode in case of eventual starvation, but it may be able to further shrink the required root of trust and thus concentrate defenders' efforts on a smaller surface area.

Not only must the RoR exist and be essentially uncompromisable, but to the greatest degree possible, everything not in the RoR must actually be recoverable. The RoR must be able to retake control of these other components, scrub their non-volatile and volatile memories, load new software and/or firmware as needed, and restart. Any vital components that can't be recovered are necessarily part of the root of trust, at a minimum, because there is no way to carry on or recover should they fail or be subverted.

6.6 MONITOR THE PRE-LAUNCH ENVIRONMENT

Satellites exit an integration facility, ready for launch, and eventually enter orbit. They pass through many months and many hands between those two points and during that time, the satellite may be vulnerable to manipulation by those entrusted to its care. A variety of anti-tamper techniques are used to protect systems that may be placed in physical danger of alteration, but these techniques may add unacceptable levels of fragility, weight, and cost to the satellite. Novel approaches may be needed.

6.7 SUCCEED QUICKLY

Recovery from an anomalous situation may require restarting or rebooting some subset of the satellite's systems. During this recovery process, the satellite is not accomplishing mission goals. If the satellite's restart process is particularly lengthy, simply trying to recover may itself jeopardize or thwart mission objectives, even if recovery succeeds.

There are periods of a satellite's life where recovery can and arguably should be very slow and very cautious. When satellites first reach orbit, they frequently undergo an extensive "check-out" period during which systems are methodically tested and baseline performance is established. It makes good sense to be very patient and careful during checkout.

During operation, however, satellite systems should be built to reboot quickly whenever possible—ideally within seconds. Systems can be designed to differentiate between a serious hardware fault (which merits a careful system check on reboot) and a commanded or autonomous reboot (which could opt to skip hardware checks in exchange for speed). As needed, various subsets of self-check could be reenabled if repeated reboots occur or if a reboot evaluates stored state data and finds them wanting.

6.8 SECURITY AS A FEATURE

Proper security measures increase the likelihood of mission success, and are therefore valuable contributions to a system's design. However, engineering pressures often prioritize functionality over security. The resulting system works, but may not work acceptably under fire. It is important to ensure that security survives managers' prioritization processes and that we field systems that can operate when we need them to—or not bother fielding them at all, lest they give a false sense of capability.

In general, security solutions are better accepted when they also improve other aspects of the design, e.g., by including a new feature. Secure Shell (SSH) has overwhelmingly taken over Telnet as the remote login technology of choice, but it did not necessarily do so because it is more secure than Telnet (although it certainly is). Instead, its rapid acceptance was also due to a feature that SSH provides—namely, effortless tunneling of X Windows connections. We should attempt to identify and highlight "dual benefit" applications of security technology whenever possible.

For example, the increased use of COTS processors on satellites has increased the risk of SEEs disrupting processing. Faults could occur as frequently as two to three faults per day, with many more possible as the feature size of the COTS parts shrinks [41]. Mitigation strategies exist for these scenarios, but often require "soft cores" in the field-programmable gate array (FPGA) and triple redundancy, causing greatly diminished performance [21]. For Zynq-based systems using the built-in advanced RISC machine (ARM) cores for high performance processing, however, there's nowhere to hide.

One common technique to mitigate SEEs, even on systems that attempt to internally mitigate them, is to include a simple rad-hard "watchdog" circuit that reboots the COTS part if the part locks up or otherwise appears to have a fault. However, watchdogs may not catch all errors, and particularly subtle or troublesome disruptions may cause cascading malfunction and spacecraft risk.

Engineers may therefore prefer approaches that make faults easier to find – a noble security goal as well—and mitigate faults by recovering functionality as quickly as possible once the watchdog triggers a reboot. TMR solutions such as [21] can do so in many cases, but not all, and higher-performance solutions using the dedicated ARM cores may require additional consideration in the way the code is compiled. The "Succeed Quickly" goal in Section 6.7 serves a valuable

security purpose, but it also improves the spacecraft's ability to function in the face of expected environmental damage.

This page intentionally left blank.

7. CHALLENGES TO SECURING SMALLSAT ARCHITECTURES

Many of the design guidelines discussed in the previous section are intended to allow the legitimate operators of a satellite to (re)assert control over the bus or payload in the presence of an active cyber attacker. This ability is clearly of interest to any attacker aiming to assert their own control over the satellite. Therefore, improper implementation of these design guidelines has the potential to completely undermine their intent.

In this section, we discuss key challenges to the secure architecture and implementation of cyber defenses on small satellites. In order to avoid platform-specific concerns (e.g., vulnerabilities of specific processors), we will concentrate on challenges that apply to securing any small satellite, regardless of specific bus configurations or payloads. This is not intended to be an exhaustive list, but rather an exemplar of the kinds of concerns that a secure architecture engenders and that satellite designers should be prepared to confront. To demonstrate how these challenges will influence the implementation of specific guidelines, we also consider a notional case study implementing an RoR on a small satellite.

7.1 ARCHITECTURAL CHALLENGES

7.1.1 Real-Time and Safety-Critical Software Systems Have Unique Constraints

Real-time software systems are those that must meet strict timing deadlines with respect to the processing of data. These applications generally govern physical systems, such as heating elements or reaction wheels. Control loops implemented in software receive sensor data about the physical system (e.g., temperature) and emit commands to an actuator (e.g., a heater) to bring the sensed state closer to the intended state.

A related class of software (frequently also real-time) is safety-critical software. Faults in a safety-critical application, such as a flight controller, can lead to mission failure and unrecoverable kinetic effects.

These requirements impose unique constraints on any attempt to secure them from cyber attack. First, any delays or overhead added to the system must still allow all real-time deadlines to be met, all of the time. Critically, any additional delay must be highly deterministic and mathematically proven to not cause deadline misses. Unpredictable added latencies, even if small (e.g., from cache misses), may lead to unrecoverable destabilization of physical systems.

Second, these software systems must be constantly available and running correctly. Many traditional cyber defenses crash a program upon detection of an attacker, effectively converting an attack on integrity or confidentiality into one on availability. This is acceptable for general-purpose software (e.g., web servers), but not for safety-critical or real-time applications. Any attempt to secure such systems must maintain the same reliability guarantees that the unsecured system has.

Third, real-time systems have a unique attack surface not shared by general-purpose software: timeliness. If an attacker can successfully delay incoming or outgoing data from a real-time system, even for a very short period of time, they may be able to cause a mission or platform kill.

For example, consider a software controller for a thermal management system. A thermometer periodically measures the temperature of a hardware subsystem and sends those data to the controller. In response, the controller actuates a heating/cooling element based on a calculation of how much heating or cooling, applied for how long, will most rapidly bring the system back to its safe operating range.

An attacker seeking to damage the system may delay either messages from the thermometer to the controller or from the controller to the heating/cooling element (e.g., via peripheral bus contention attacks). The system will begin oscillating between increasingly extreme temperatures. This is due to the controller inadvertently overcompensating, either because it is acting on old information from the thermometer or because its commands are not promptly acted on by the heating/cooling element. System designers are thus presented with the challenge of isolating real-time components from such attacks,while still meeting SWaP constraints.

7.1.2 Attacks Exploit Economies of Scale

The increasingly common use of COTS hardware and software components brings a phenomenon to the satellite world that attackers have leveraged for years in the world of general-purpose computing: economies of scale. It is common for a swarm of satellites to all run the same software and use the same hardware, especially if they share a common bus. Unfortunately, this also means that the entire swarm shares the same set of hardware/software vulnerabilities. If an attacker can exploit one of these, they can attack the entire swarm (albeit not necessarily simultaneously) for almost the same cost as attacking a single satellite.

This problem is exacerbated by the common reuse of a small number of operating systems across many satellites and missions (e.g., Linux, FreeRTOS). Vulnerabilities in these operating systems can let attackers compromise multiple bus configurations, payloads, and missions due to their widespread deployment. As operating systems typically operate in a very privileged mode, attackers are highly motivated to find vulnerabilities in such systems.

As a result, designers attempting to secure their satellite are presented with a challenge: how can one maintain equivalent functionality of software across multiple installations while still disrupting attacker economies of scale? Previous approaches have included mathematically proving the absence of vulnerabilities and automatically transforming the low-level representation of code used in exploitation while maintaining its high-level semantics. The former prevents certain classes of exploitation at all, while the latter ensures that a vulnerability on one system cannot leveraged to attack another. However, proving the absence of bugs has traditionally been extremely labor-intensive and has not been scaled to large, complex systems. (In Appendix A, we nevertheless adopt this approach for key components.) Low-level code transformations do scale, but may interfere with real-time constraints or provide more limited security guarantees. In general, solutions developed for the world of general-purpose computing should be seen as a starting point, but may not be directly applicable to the space domain without substantial adaptation.

7.1.3 Attacks Do Not Obey Abstractions

Modern computer languages do not provide developers with direct access to hardware as part of the language model. Rather, they work in concert with the operating system to provide an abstract model of a computer against which programs can be written. Abstraction is useful, as it can eliminate many common kinds of bugs (e.g., memory corruption in memory-safe languages). Even low-level languages such as C provide some degree of abstraction. Memory in C is treated as something directly accessible rather than the reality of memory as a subsystem interacted with through a complex array of microprocessors and buses. In many applications, the operating system also provides the abstraction of virtual memory. This isolates processes in separate memory regions, and allows the operating system to enforce permissions and constrain the behavior of potentially buggy or malicious applications.

Attacks, however, do not need to obey the abstractions offered by programming languages. A designer attempting mitigate some class of vulnerability (e.g., memory corruption) must consider the fact that attackers may be able to bypass the defense entirely by acting outside of the machine model presented to legitimate developers.

In the memory corruption domain, for example, an attacker could corrupt an otherwise memory-safe program by accessing memory outside of the channels monitored by operating system. One avenue that has been leveraged in the past is direct memory access (DMA). DMA is a technique used by high-bandwidth peripherals (e.g., network, storage, and video devices) to read/write memory directly, without requesting the operating system to do so on their behalf (as is the case with conventional user-space applications). This enables improved throughput, but could also let an attacker bypass memory permissions and process isolation if they can induce a DMA-capable peripheral to operate on their behalf. Other options available to the attacker include architectural side-channels such as speculative execution vulnerabilities [42] and hardware bugs such as Rowhammer [43], all of which act outside the abstractions presented by the language and operating system.

A similar challenge arises with respect to protection of data in transit. Despite the abstractions afforded by the language and operating system, an application is not running in isolation on a machine. Many other applications are simultaneously executing, some of which may be executing on separate processors on peripheral devices from the main CPU. If encryption/decryption is done when data leaves/enters the computer system, it may be passing in plaintext over internal hardware buses connecting the communications system to the target process. Malicious processes with access to these buses could intercept or modify such messages despite the perception of security afforded by the cryptography.

Even if encryption is truly end-to-end, malicious processes may still be able to disrupt the target process. Denial-of-service attacks on shared communications buses can cause message loss and prevent critical commands from being executed. Alternatively, data, including key material, may be vulnerable to exfiltration via side-channels in shared hardware (e.g., timing of executed instructions).

Therefore, in order to securely implement a cyber defense, designers must be aware of the larger context that defense operates in, especially with respect to the lower-level system components whose operational details are abstracted away. In order to actually provide the guarantees they hope to, it may be necessary to implement defenses working at multiple levels of abstraction.

7.1.4 Shared Resources Amplify Attack Impact

Software designers for SWaP-constrained environments must make as efficient use of resources as possible. This frequently leads to a software base with substantial resource sharing and accessibility. For example, NASA's Core Flight System (cFS) is (as of 2018) implemented as a single process, with each service running in its own thread. This enables common state to be shared among services rather than needing to be duplicated and taking up more memory. (We address this specific issue in Appendix A.1.)

Unfortunately, resource sharing also provides opportunities to attackers to further cement their hold on a system or cause denial of service. Consider the cFS mentioned above. If any individual service is compromised by an attacker, all other services are now at risk due to their shared address space. More generally, any resource shared between a compromised component and a target component is an attack surface that must be considered and mitigated. This includes not only shared memory, but also the operating system managing bus and/or payload software, the hardware executing that software, and any hardware peripherals that could modify system state (e.g., main memory or data on a bus).

This poses a challenge to system architects. Ideally, all software processes should be as isolated from one another as possible, with minimal ability to affect one another outside of carefully monitored channels. However, this is not only difficult in SWaP-constrained environments, it is often physically impossible due to a shared hardware substrate. That substrate may be able to be leveraged by an attacker to leak information (e.g., side channels), deny availability (e.g., bus contention), induce kinetic effects (e.g., firing of thrusters), and so on. Since total isolation is often unachievable, it should be seen as a necessary, but not sufficient, component of a secure architecture.

Another challenge to shared resource access is that the need to access a resource is not static, and may change over time based on a variety of factors (e.g., satellite operating mode). For example, if a hardware or software component is compromised and the attack is detected, it should be quarantined and unable to access the resources that the uncompromised system. Even absent a detected attack, sensitive resources that are only accessed in exceptional circumstances (e.g., thrusters) should only be accessible by software during such exceptional circumstances, in order to minimize what an undetected attack could achieve. Therefore, system architects must consider how resource access by bus and payload software can be dynamically adjusted over time without impacting the ability of the satellite to conduct its mission.

7.2 CHALLENGES IN BUILDING A SECURE ROOT OF RECOVERY

The challenges enumerated above are deliberately general and not tied to a specific design guideline from Section 6. In this section, we demonstrate how those general challenges distill into

challenges specific to a particular guideline. The RoR described in Section 6.5 is a natural target for attackers. If built incorrectly, it may serve as highly privileged backdoor into the satellite that allows an attacker to upload and execute malicious code. Securing it is paramount, and must go beyond simple software testing.

Although the exact architecture and implementation may be platform-specific, any RoR implementation will likely include a combination of software (an updater capable of receiving, storing, and loading a program image), hardware that that software depends on (including the radio, embedded power supply, etc.), and the firmware drivers supporting that hardware. All of these must be architected in a secure way in order to protect the RoR from use by an adversary. This imposes several platform-independent challenges on satellite designers. More challenges will arise when considering platform-specific details. The following are intended as examplars, not an exhaustive list.

7.2.1 Attackers Will Try to Persist Across Software Updates

An attack may persist even if the malware used to launch that attack has been purged. For example, an attacker may corrupt stored data that will cause a legitimate process to make decisions chosen by the attacker, or may spread to an accessible resource or other program. The designer must ensure that all dataflows out of a compromised process, at all levels of abstraction, are within the scope of a recovery operation.

7.2.2 Malicious Software and Hardware Will Try to Corrupt or Disable the Root of Recovery

The RoR is uniquely able to overwrite the stored program image of every software application on the satellite. Depending on platform and implementation details, it may also be able to write to the memory image of every running process (e.g., to support live patching). Simultaneously, it must itself be immutable and impervious to an already-compromised applications attempt to corrupt it. Thus, designers must consider how to architect the system so that the RoR can securely write to every process, while not being writable by any process.

In addition to isolation from malicious software, malicious hardware (e.g., introduced during the supply chain) may attempt to disrupt or disable the RoR. This includes not only corruption of the program, but also denial of service via attacks on shared resources such as hardware buses. Designers must consider how to isolate the RoR from malicious hardware while still meeting SWaP constraints.

7.2.3 Remote Attackers will Target the RoR

By definition, the RoR must accept input from the ground station (i.e., the software image), parse that input (even prior to decryption), and process it. Input handling logic is notoriously difficult to write, and in many languages involves bug-prone operations such as casting memory as a user-specified type, copying to/from/between buffers, manipulating pointers, and so on. Attackers are well aware of this and will likely target the RoR with maliciously formatted inputs. Any

successful exploitation of a bug on the RoR will have devastating impacts on the security of the satellite and the ability of the legitimate operator to re-assert control.

As a result, designers must seek to make RoR code as highly assured as possible. Ideally, the code should be formally verified to be free of security-critical bugs such as memory corruption. Lacking the time or budget to do so, designers should find ways to leverage memory-safe languages that provide compile-time guarantees about bug-freeness.

In the unfortunate event that an attack on the RoR succeeds, attackers will try to exploit economies of scale and leverage that attack against other satellites. Designers must seek to disrupt this asymmetric advantage. RoR code should not afford attackers economies of scale and should be diversified at compile time, randomized (with respect to memory layout) at run-time, or otherwise made unique such that an attack on one satellite cannot become an attack on all.

7.2.4 Attackers May Target the Update Authorization Mechanism

The cryptographic key to communicate with the satellite should not be the same key used to sign software updates, as these are much more sensitive than other operations. Should attackers obtain the key material to enable communication, they can attempt to exploit bugs in any authorization mechanism used by the RoR to determine whether an update is from a trusted source. Designers must therefore have highly assured authorization verification mechanisms, ideally a formally verified algorithm for validating cryptographic assurances on signed updates.

7.2.5 The RoR Cannot Violate Realtime Constraints

The purpose of the RoR is to allow remote code updates to satellite subsystems. Many of these systems are safety-critical and/or real-time and must therefore adhere to stringent timing requirements for message processing. This presents designers with two challenges to implementing a safe RoR.

First, they must consider how to handle intermediate program state (e.g., values of variables on the stack) at the time of update/reboot. If purging this state would result in a fault or real-time violation, the RoR must have some mechanism to securely mitigate or prevent such a fault, while also ensuring that an attacker cannot cause a persistent attack by leveraging this state information.

Second, systems being modified by the RoR need to continue meeting real-time requirements during update/reboot. Periodic processing deadlines (e.g., from a sensor feed) are often specified in terms of microseconds, while software update/reboot operations could range from milliseconds to seconds. Loss of availability during an update could result in unrecoverable kinetic effects and so must be avoided in any safe RoR implementation.

8. CONCLUSION

Satellite design already demands a substantial and diverse set of engineering disciplines, and the evolving ecosystem of small satellite developers, vendors, and buyers are applying increased pressure on the design timelines for these systems. The need to protect against benign faults and the threats of the hostile orbital environment consume engineers' attention, and security can fall by the wayside.

We do not seek to solve that problem in the general sense. There will be a need for "standard" security engineering practices, and their evolution continues on the ground and in space. However, we posit that a small set of guidelines and design principles, carefully tailored for the space environment, can further improve the efficiency of satellite design and the efficacy of the resulting system. It remains for us to prove it by example, and we fully intend to do so in our continuing work.

This page intentionally left blank.

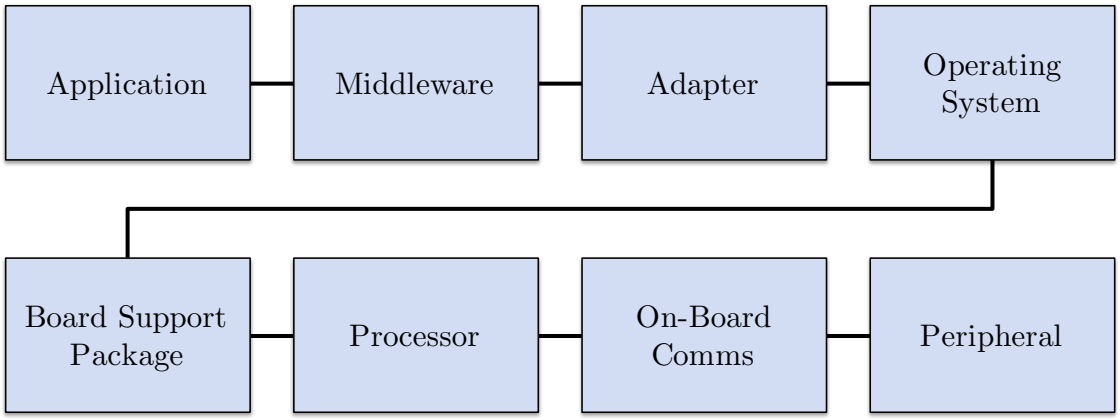

Figure A.1. Processing flow for a notional bus design.

APPENDIX A: ARCHITECTURE SKETCH

We intend to realize our guidelines in the form of a sample satellite processing architecture that could provide a useful starting point for a variety of new designs. The basic layout of the processing flow for our notional bus design is shown in Figure A.1.

Our goal is to provide a familiar model at the ends of the stack (application and peripheral) while fostering an improved security foundation in the middle of the stack. Doing so would allow us to provide the key underpinnings for the RoR guideline and provide a suitable foundation for system designers to pursue the rest of the guidelines.

As a representative middleware-and-adapter stack, we have chosen the NASA Goddard cFS. The cFS suite is already in use for several missions, so designers are familiar with it. cFS includes an operating system abstraction layer (OSAL) adapter that allows it to run on a variety of full-featured operating systems, namely POSIX[4], FreeRTOS, and VxWorks.

We leverage the increasingly ubiquitous Xilinx Zynq System-on-Chip as the processor. The Zynq is a very powerful device, featuring two ARM CPUs and a generous amount of FPGA reconfigurable logic. It is far more powerful than the more traditional RAD750 processor. Although it is not suitable for extended mission durations or higher orbits, it is being used in LEO designs. We anticipate that future rad-hard processors will also target the ARM architecture.

The Zynq is packaged into boards suitable for space use. For our testing, we employ a simple ZedBoard test fixture. For a fielded system, something like the CHREC[5] Space Processor (CSP) would be appropriate. The CSP adds a watchdog circuit and other necessary hardware to handle the

[4] POSIX is not an operating system; rather, it's a set of APIs that several operating systems support. Practically speaking, NASA uses this for varieties of the Linux OS.

[5] CHREC is now named SHREC, but the name of the processor remains unchanged at this time.

SEEs that occur in orbit. CHREC continues to develop and integrate additional firmware and software to this end, including techniques for mitigating SEEs in the FPGA logic itself. The CSP is being sold as a commercial product by at least one vendor, so there is a path to fielding for systems based on this board.

There are a variety of onboard communications buses in use on satellites, and the variety increases all the time. For our initial instantiation, we are targeting the Inter-Integrated Circuit Bus, or I^2C bus, as it is of representative speed and complexity.

The final pieces of the puzzle are the operating system and the board support package. To support the RoR guideline, we need to build the core of the system on a foundation of utmost security. For this purpose, we have opted for the seL4 microkernel [44]. seL4 has been formally proven to faithfully implement its specification. This means that the microkernel is proven to have no buffer overflows, deadlocks, or livelocks, among other undesirable conditions. That isn't perfection, but it's far closer to enlightenment than its informally architected brethren.

Sections A.1 through A.3 will describe three key hurdles that we overcame in our effort to provide the NASA cFS software stack on top of the seL4 microkernel instead of the feature-rich kernels upon which it is typically employed. Section A.4 will describe next steps for our specific implementation, and Section A.5 will conclude with next steps for a system implementation on top of the seL4 + cFS foundation.

A.1 PREPARING CFS FOR SEL4: PROCESSES

As described in Section 7.1.4, the NASA cFS employs multiple simultaneously executing agents. The OSAL provides a `taskCreate` method to spawn additional concurrent tasks. At present and across all supported operating systems, this method creates threads, not processes. As a result, any cFS task has complete read/write access to all memory belonging to all tasks, as shown in Figure A.2a. cFS takes advantage of this access to share state via an extensive collection of global variables.

NASA has tested cFS extensively, and indeed, spacecraft engineers often embark on extensive testing regimes. Although that may give good assurances of safety, it does not necessarily warrant any confidence in the system's security.

We have created a patch for cFS that, on POSIX, modifies `taskCreate` to create separate processes, rather than threads. As part of this switchover, we adjusted the inter-task communication mechanism to provide cross-process access. We additionally identified all global variables, isolated them in a shared-memory region, and then provided that region and associated pointers to all cFS processes.

The result, shown in Figure A.2b, is that a compromised or corrupted task can still damage arbitrary amounts of global data, but it can no longer read or write the private data of other tasks. This reduces the attack surface of an individual task and also arguably improves safety, as a bug is now unable to damage task-private state.

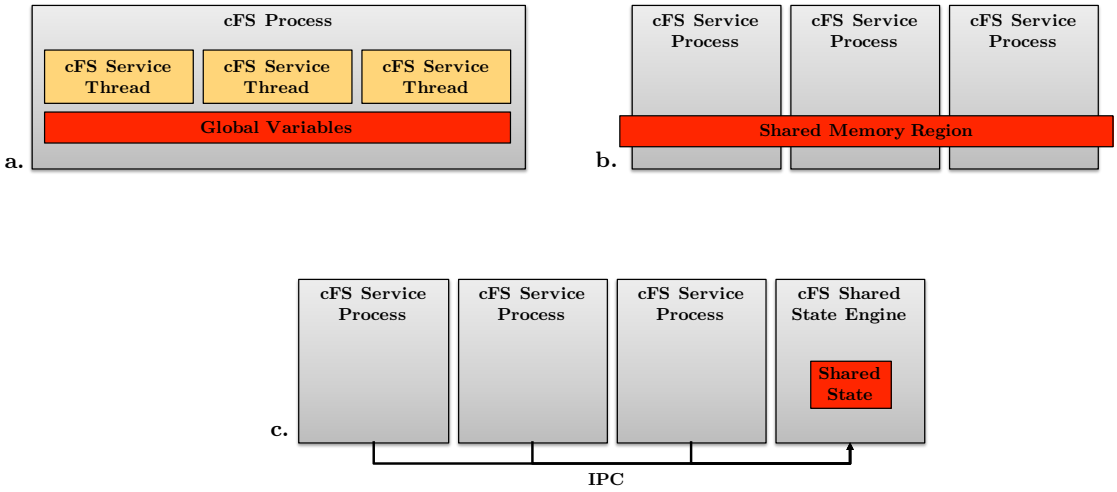

Figure A.2. Memory structures: (a.) thread-based, (b.) process-based, (c.) getter–setter based.

This work aids us in our effort to move cFS onto seL4, as the seL4 microkernel is better suited to process-based isolation. However, the shift from threads to processes in this case should also improve security across all operating systems.

In the future, it would be better still to implement proper access control on the global variables, as illustrated in Figure A.2c. If individual processes had access only to getter and setter functions instead of raw access to global state, then the opportunities for both mistakes and for compromise are reduced. Such an approach also provides the groundwork for the dynamic isolation suggested in Section 7.1.4.

A.2 INTERFACING I^2C WITH SEL4

To connect the seL4 operating system to the satellite's peripherals, as shown on the right-hand side of Figure A.1, we developed an example device driver for the combination of the seL4 kernel and the Zynq system-on-chip. Our initial effort, focusing on the I^2C bus, provides sufficient realism to prove that a reasonable path forward exists.

In seL4, all device drivers are processes, just like any other process. The best mechanism available to isolate device access to a single process is to use memory-mapped input and output (MMIO) to the device in question and then limit access to that memory-mapped region to only the device driver's process. The driver can then provide inter-process communication (IPC) calls for other processes to use to actuate the device, and proper access control can be set up to ensure that only authorized control processes can access the driver and thus the device.

Implementing this in practice requires a careful synchronized dance between seL4, the Zynq part, and the Xilinx Vivado development environment with which the Zynq is programmed. The Vivado environment is used to autogenerate several pieces of the puzzle:

Figure A.3. "First light" communicating via I²C on seL4.

1. An I²C physical layer (PHY) implementation, in the FPGA fabric
2. A set of I²C functions in C, for use within the Zynq's ARM cores
3. The necessary Advanced eXtensible Interface (AXI) bus configurations and MMIO regions to bridge the FPGA physical interface layer (PHY) to the ARM-based functions

The Xilinx code, intended for use on a more feature-rich operating system, is nevertheless amenable to seL4. The key is connecting the Vivado dots appropriately. The Vivado file `system.hdf` contains the base and limit addresses for our MMIO interfaces. Those values need to be carried over to the seL4 driver process.

Within seL4, the function `sel4platsupport_new_io_mapper()` is used to create a structure for performing the mapping. The function `ps_io_map` can then be used to map the MMIO's physical address range into the virtual address space of the driver process.

To test the driver concept, we connected a ZedBoard development board to a PC and drove the I²C bus from the PC to emulate a target device. The resulting "Hello World" exchange, shown in Figure A.3, is visually underwhelming, but represents an important first step.

To further test the driver concept, we used an Arduino board provided by NASA IV&V. The Arduino was programmed to emulate a power supply subsystem by Clyde Space. We were able to connect the Arduino to the same ZedBoard and perform basic C2 of the subsystem from seL4.

Our goal was to simply prove the concept. Future work could easily incorporate other bus types (e.g., SPI), increase the realism, and build out APIs by which applications could command the drivers.

A.3 PORTING THE CFS OSAL TO SEL4

The cFS's OSAL contains roughly 100 operating system-agnostic calls. They are grouped into the following categories:

- Miscellaneous (time management, debug printing)
- Message queues
- Timers
- Semaphores and mutexes
- Networking
- File system (generic Network File System-like methods)
- Interrupts
- Exceptions

To properly emulate cFS on seL4, the entirety of the OSAL API needs to be emulated. Some portions of the API are reasonably tractable—notably, the file system can be emulated in a fairly generic fashion—but other aspects are more intimately tied to operating system features. For our proof of concept, we opted to focus on aspects of the semaphore API, as synchronization is something that the operating system needs to supply.

Implementing mutexes and semaphores on seL4 is nontrivial. As seL4 provides isolation, IPC, and processes, in the seL4 model. a mutex or semaphore is effectively a process that offers the `lock` and `unlock` operations. Rather than construct this from scratch, we leverage a seL4 toolset called caMKes, or component architecture for microkernel-based embedded systems. caMKes—and the mutex primitive it provides—has the added benefit of being formally verified, so the code it generates carries similar guarantees to that of the kernel upon which it executes.

The one caveat is that caMKes is statically focused; it's designed for systems that know at compile-time which mutexes exist and to which processes they are tied. This directly conflicts with the cFS runtime model for mutexes, wherein they are named and established in a dynamic fashion. For example, see Listing 1, which shows how mutexes are created in the OSAL POSIX

layer. Note that under the hood, it dynamically creates an ordinary POSIX mutex and assigns it to a dynamically provided name `sem_name`. Under the hood, OSAL maintains a statically allocated array of potential mutexes and "slots" a given mutex into that array, indexed by C string name, for later lookup in calls that use the semaphore.

This contradicts the caMKes model—OSAL is dynamic, and caMKes is static. We work around this difference in a reasonably straightforward manner: we instrument the OSAL methods related to semaphore usage, execute the test suites for a satellite's control software on a POSIX OS, and record the creation and usage patterns for the application's semaphores. Given those patterns, we are able to statically create (via caMKes) everything that the application will dynamically request and satisfy the (dynamic) API using seL4's established (static) mutex techniques.

This method—dynamic instrumentation to provide static resources—is in general a poor approach; if the dynamic testing is not exhaustive, then an on-orbit corner case could yield a mutex operation for which the statically allocated resources are not prepared. However, satellite designers are typically extremely rigorous in their testing. We are relying on that property to ensure that the dynamic instrumentation approach is sufficient to garner correct and complete coverage.

Listing 1: OSAL `OS_BinSemCreate` for POSIX

```
int32 OS_BinSemCreate (uint32 *sem_id, const char *sem_name,
                       uint32 sem_initial_value,
                       uint32 options)
{
    int                  Status;
    pthread_mutexattr_t  mutex_attr;
    pthread_mutex_t      mutex;
    pthread_cond_t       cond;
    /* ... */
    Status = pthread_mutexattr_init(&mutex_attr);
    Status = pthread_mutexattr_setprotocol(&mutex_attr,
                    PTHREAD_PRIO_INHERIT);
    Status = pthread_mutex_init(&mutex, &mutex_attr);
    Status = pthread_cond_init(&cond, NULL);
    /* ... */
}
```

A.4 NEXT STEPS FOR THE ROOT OF RECOVERY

The work described in Section A.1 through Section A.3 work to establish some of the foundation for an eventual root of recovery (RoR) for satellite systems. Three challenges remain: completion of the OSAL, construction of the RoR on the processing element, and establishment of the RoR on critical subsystems.

First, the OSAL has a large repertoire of features beyond the semaphores we have thus far implemented. Intelligent Automation has built a "POSIX-like" layer [45] that could be leveraged

to more easily complete the OSAL layer on seL4. Evaluation of this layer's suitability and implementation of same remain as future tasks.

Second, the work described thus far only provides the foundation for an actual RoR. A true RoR, built atop seL4, must be able to either reboot to a known prior state or to receive a new state (software update) from ground control and reboot to it. The software update mechanism itself must be well understood and invulnerable, lest a flaw in the update mechanism cause spacecraft loss.

Lastly, as observed in Section 7.2, the RoR comprises more than just the operating system and associated processes. Indeed, the RoR relies on everything used to command the recovery process. This includes inline cryptographic processing, the radio by which the recovery command is received, and the Flash memory and power control components used to re-image and reboot the remainder of the satellite. Any or all of these components may be software-defined, which further blossoms the RoR's scope and influence. In this work, we neglect consideration of a software-defined radio's software and firmware; in a complete system, it must be within scope and fully considered.

These are substantial challenges. It is work enough to build them at all and it is substantially more work to build them to the formal verification standards to which seL4 is held. Yet this is work that must be undertaken in order to ensure our satellites are recoverable should something befall them. Such a scheme provides benefits absent an adversary as well, of course; coupled with a verified software update mechanism and a "root of survival" that sees to the spacecraft's basic power, communications, and thermal needs no matter what, the satellite's owner can be far freer with new and experimental software loads knowing that the satellite can be recovered if they go awry.

A.5 NEXT STEPS FOR AN ADOPTING SATELLITE

All of the above work has operated on pieces of the puzzle in isolation, but a full proof of suitability requires a more extensive satellite environment. Ideally, we'd demonstrate a seL4-based cFS environment, with an application, driving a bare-bones set of representative satellite peripherals (radio, power, etc.) within the context of a simple, realistic physics model and ground system.

To achieve those goals, we would turn to the NASA NOS3 suite and STF-1 test satellite [46]. NOS3 includes environmental modeling, orbit modeling, a sample ground station, and robust models of various satellite infrastructure (a GPS receiver and a radio, among others). The NOS3 suite ensures that we can interact with representative satellite hardware and establish a cost-effective testbed for further development and demonstration.

At present, the NOS3 suite is not able to emulate non-Linux spacecraft operating systems. This is an artifact of how NOS3 currently models peripherals; they are instantiated as Linux libraries rather than as true virtual peripherals. The NASA IV&V team that maintains NOS3 is interested in supporting diverse spacecraft operating systems, as NASA does leverage the non-Linux operating systems that cFS supports. We therefore anticipate that future versions of NOS3 will easily support a seL4-based environment as well.

On top of this design, we intend to honor the basic mitigation strategies described in Section 2.4.3 as well as our design guidelines.

Securely Parse and Ingest Data. Satellite C2 traditionally operates in a fairly data-starved environment, with limited bandwidth and minimal interactivity. The data are therefore reasonably easy to restrict to a simple set for ingestion, largely dictated by well-worn CCSDS standards. The ingestion and validation of the commands and their parameters may still be challenging, however.

Surface/Component Minimization. Satellite designers have traditionally done a good job here; the challenge is going to be maintaining that parsimony in the new land of plenty afforded by the Zynq's capabilities.

Component Isolation. The "system-facing" interface is minimal. seL4 has only a few function calls, and we're highly confident of its correct operation. Each "customer-facing" interface (device driver) can be separated into individual seL4 tasks for effective isolation.

Data Protection (At Rest, In Transit, In Use). We will leverage the MIT LL LOCKMA library to perform cryptographic operations as needed for data at rest and in transit. We rely on seL4 for compartmentalization of data in use.

Authentication and Secure Control. Both terminals and C2 should have strong access control. This principle requires fine-grained access control; for example, a terminal should specify who can send data, how much, and to whom.

Randomize, Diversify, and Adapt. We are unlikely to pursue dynamic approaches here, even the more common approaches such as address space layout randomization (ASLR), in our initial version. We could perhaps investigate its inclusion in future work, but it would be challenging to implement without touching the operating system. Other automatic diversity techniques such as multi-compilers could be leveraged as well, especially if custom compilers are already being used for other purposes (such as software-based insertion triple modular redundancy [TMR] in the application code) and the customizations are compatible (e.g., cascading LLVM intermediate representation [IR] modifications).

Rapid Replacement and Reconfiguration. Traditionally, satellite operators do extensive testing before applying a software update on orbit due to the danger involved and the risk of an unrecoverable mistake. The "safe" software update scheme we envision should reduce the risk of patching and thus reduce the time necessary to validate a patch before deployment.

Our initial work does not address the size of the patch. Replacing the entire boot image is a painful proposition due to the typically low C2 data rates. Future work should address more bandwidth-efficient incremental patching.

Not every problem is a software problem; the problem could also be corrupted or invalid data being processed by otherwise adequate software. The RoR abstraction enables efficient and effective scrubbing and reconstitution of local data storage as well.

This page intentionally left blank.

APPENDIX B: CREF TABLE

Adaptive Response	Respond appropriately and dynamically to specific situations using agile and alternative operational contingencies to maintain minimum operational capabilities in order to limit consequences and avoid destabilization, taking preemptive action where appropriate
Analytic Monitoring	Continuously gather, fuse, and analyze data to use threat intelligence, identify vulnerabilities, find indications of potential adverse conditions, and identify potential or actual damage
Coordinated Defense	Coordinate multiple, distinct mechanisms (defense-in-depth) to protect critical resources across subsystems, layers, systems, and organizations
Deception	Confuse, deceive, and mislead the adversary
Diversity	Use a heterogeneous set of technologies, data sources, processing locations, and communications paths to minimize common mode failures (including attacks exploiting common vulnerabilities)
Dynamic Positioning	Distribute and dynamically relocate functionality and assets
Dynamic Representation	Support mission situation awareness and response by using dynamic representations of components, systems, services, adversary activities and other adverse situations, and the effects of alternative courses of action
Non-Persistence	Retain information, services, and connectivity for a limited time, thereby reducing exposure to corruption, modification, or usurpation
Privilege Restriction	Design to restrict privileges assigned to users and cyber entities and to set privilege requirements on resources based on criticality
Realignment	Enable resources to be aligned (or realigned) with core mission functions, thus reducing the attack surface, the potential for unintended consequences, and the potential for cascading failures
Redundancy	Provide multiple protected instances of critical information and resources, to reduce the consequences of loss
Segmentation/Separation	Separate (logically or physically) components based on criticality and trustworthiness to limit the spread of damage
Substantiated Integrity	Provide mechanisms to ascertain whether critical services, information stores, information streams, and components have been corrupted
Unpredictability	Make changes frequently and randomly to make the attack surface unpredictable

TABLE B.1

Cyber Resiliency Techniques from Table 3 of [3]

GLOSSARY

AFSCN	Air Force Space Control Network
C2	Command and Control
CAPEC	Common Attack Pattern Enumeration and Classification
CCSDS	The Consultative Committee for Space Data Systems
CNSS	Committee on National Security Systems
CNSSI	Committee on National Security Systems Instruction
CONOPS	Concept of Operations
CONUS	Contiguous United States
COTS	Commercial Off-The-Shelf
CREF	Cyber Resilience Engineering Framework
DSB	Defense Science Board
FIPS	Federal Information Processing Standards
FPGA	Field-Programmable Gate Array
GEO	Geosynchronous Orbit
GTO	Geosynchronous Transfer Orbit
HEO	Highly Elliptical Orbit
ISR	Intelligence, Surveillance, and Reconnaissance
LEO	Low Earth Orbit
LL	Lincoln Laboratory
MIT	Massachusetts Institute of Technology
NIST	National Institute of Standards and Technology
POC	Payload Operations Center
RBAC	Role-Based Access Control
RF	Radio Frequency
RMF	Risk Management Framework
RMF	Risk Management Framework
RoR	Root of Recovery
RTG	Radioisotope Thermoelectric Generator
SEE	Single Event Effects

GLOSSARY
(Continued)

SOC	Spacecraft Operations Center
SoC	System-on-Chip
STAMP	Systems-Theoretic Accident Model and Processes
STECA	System-Theoretic Early Concept Analysis
STPA	Systems-Theoretic Process Analysis
SV	Space Vehicle
SWaP	Size, Weight, and Power
TID	Total Ionizing Dose
TRANSEC	Transmission Security
TT&C	Telemetry, Tracking, and Command
WGS	Wideband Global SATCOM

REFERENCES

[1] P. Kaminski (ed.), *Task Force Report: Resilient Military Systems and the Advanced Cyber Threat*, Washington, DC 20301-3140: Office of the Under Secretary of Defense for Acquisition, Technology and Logistics (2013).

[2] D.A. Plunkett (ed.), *CNSSI No. 1253F Attachment 2: Space Platform Overlay*, Committee on National Security Systems (2013).

[3] D. Bodeau and R. Graubart, "Cyber Resiliency and NIST Special Publication 800-53 Rev.4 Controls," MITRE, Technical rep. (2013).

[4] L.M. Anderson, B. Raynor, and M. Hurley, "TacSat-4: Military Utility in a Small Communication Satellite," in *27th Annual AIAA/USU Conference on Small Satellites* (2013).

[5] N.A. Dallmann, J.G. Delapp, D.C. Enemark, T.D. Fairbanks, C.M. Fortgang, D.C. Guenther, S.L. Judd, G.M. Kestell, J.E. Lake, J.P. Martinez, K.P. McCabe, J.M. Michel, J.M. Palmer, M.W. Powell, D.A. Prichard, M.C. Proicou, H.M. Quinn, R.S. Reid, E.B. Schaller, D.N. Seitz, P.S. Stein, S.A. Storms, E.A. Sullivan, J.L. Tripp, A. Warniment, and R.M. Wheat, "An Agile Space Paradigm and the Prometheus CubeSat System," in *29th Annual AIAA/USU Conference on Small Satellites* (2015).

[6] D.M. Maybury, "Technology Horizons: A Vision for Air Force Science and Technology 2010-2030," Office of the US Air Force Chief Scientist, Technical rep. (2010).

[7] E.S. Nightingale, L.M. Pratt, and A. Balakrishnan, "The CubeSat Ecosystem: Examining the Launch Niche," Institute for Defense Analysis, Technical Rep. IDA Document NS D-5678 (2015).

[8] URL http://www.cubesat.org/suppliers/.

[9] URL https://capec.mitre.org/.

[10] Joint Task Force Transformation Initiative, "Guide for Applying the Risk Management Framework to Federal Information Systems," National Institute of Standards and Technology, Technical rep. (2010).

[11] Joint Task Force Transformation Initiative Interagency Working Group, "NIST Special Publication 800-53 Revision 4: Security and Privacy Controls for Federal Information Systems and Organizations," National Institute of Standards and Technology, Technical rep. (2013), URL dx.doi.org/10.6028/NIST.SP.800-53r4.

[12] "FIPS PUB 199: Standards for Security Categorization of Federal Information and Information Systems," National Institute of Standards and Technology, Technical rep. (2004).

[13] N.G. Leveson, "An STPA Primer, Version 1, August 2013 (updated June 2015)," URL http://psas.scripts.mit.edu/home/wp-content/uploads/2015/06/STPA-Primer-v1.pdf.

[14] N.G. Leveson, "A New Accident Model for Engineering Safer Systems," in *Safety Science* (2004), vol. 42, pp. 237–270.

[15] N.G. Leveson, *Engineering a Safer World: Systems Thinking Applied to Safety*, MIT Press (2012), URL https://mitpress.mit.edu/books/engineering-safer-world.

[16] C.H. Fleming and N. Leveson, "Integrating Systems Safety into Systems Engineering during Concept Development," in *25th Annual INCOSE International Symposium* (2015).

[17] J. William E. Young, "STPA-SEC for Cyber Security / Mission Assurance," (2014), URL psas.scripts.mit.edu/home/wp.../Young_STAMP_2014_As-delivered.pdf.

[18] J. William E. Young and N.G. Leveson, "An Integrated Approach to Safety and Security Based on Systems Theory," *Communications of the ACM* 57(2) (2014).

[19] D. Byrne, "Cyber-Attack Methods, Why They Work On Us, and What To Do," in *AIAA SPACE Conference and Exhibition* (2015).

[20] K.L. Bedingfield, R.D. Leach, and M.B. Alexander, "NASA Reference Publication 1390: Spacecraft System Failures and Anomalies Attributed to the Natural Space Environment," NASA, Technical rep. (2013).

[21] D. Sinclair and J. Dyer, "Radiation Effects and COTS Parts in SmallSats," in *27th Annual AIAA/USU Conference on Small Satellites* (2013).

[22] A. Witze, "Software error doomed japanese hitomi spacecraft," *Nature* 533(7601), 18–19 (2016), URL http://www.nature.com/news/software-error-doomed-japanese-hitomi-spacecraft-1.19835.

[23] "Saving NASA's STEREO-B: The 189 Million Mile Road to Recovery," URL http://www.nasa.gov/feature/goddard/saving-nasas-stereo-b-the-189-million-mile-road-to-recovery.

[24] J.W. Dankanich, "Small Satellite Propulsion," NASA, Tempe, AZ, Technical rep. (2015), Presentation at Conference on Spacecraft Reconnaissance of Asteroid and Comet Interiors (AstroRecon); 8-10 Jan. 2015.

[25] W.W. Ward, "Developing, Testing, and Operating Lincoln Experimental SSatellite 8 and 9 (LES-8/9)," MIT Lincoln Laboratory, Technical Rep. Technical Note 1979-3 (1979).

[26] R.W. Berger, D. Bayles, R. Brown, S. Doyle, A. Kazemzadeh, K. Knowles, D. Moser, J. Rodgers, B. Saari, D. Stanley, et al., "The RAD750™ a radiation hardened PowerPC™ processor for high performance spaceborne applications," in *Aerospace Conference, 2001, IEEE Proceedings.*, IEEE (2001), vol. 5, pp. 2263–2272.

[27] "Cute-1.7 + APD II Project – Sub Systems and C&DH," URL http://lss.mes.titech.ac.jp/ssp/cute1.7/subsystem_cdh_e.html.

[28] J. Lyke, Q. Young, J. Christensen, and D. Anderson, "Lessons Learned: Our Decade in Plug-and-Play for Spacecraft," in *28th Annual AIAA/USU Conference on Small Satellites* (2014).

[29] M. Martin and J. Lyke, "Modular Open Network ARCHitecture (MONARCH): Transitioning Plug-and-Play to Aerospace," in *Aerospace Conference, 2013 IEEE* (2013).

[30] S.H. Schaire, S. Altunc, and B.K. Malphrus, "Cubesat Communication Direction and Capabilities At Morehead State University and NASA Goddard Space Flight Center, Wallops Flight Facility," in *27th Annual AIAA/USU Conference on Small Satellites* (2013).

[31] D. Caplan, J. Carney, R. Lafon, and M. Stevens, "Design of a 40 Watt 1.55 μm Uplink Transmitter for Lunar Laser Communications," in *SPIE LASE*, International Society for Optics and Photonics (2012), pp. 82460M–82460M.

[32] F.C. Belz, "Space Mission Resilience to Cyber Attacks," The Aerospace Corporation, Technical rep. (2012).

[33] Consulative Committee for Space Data Systems (CCSDS), *Security Threats Against Space Missions*, 350.1-G-2, CCSDS (2015).

[34] Consulative Committee for Space Data Systems (CCSDS), *Security Architecture for Space Data Systems*, 351.0-M-1, CCSDS (2012).

[35] Consulative Committee for Space Data Systems (CCSDS), *CCSDS Report Concerning Space Missions Key Management Concept*, 350.6-G-1, CCSDS (2011).

[36] A. Zeliff, C. Roman, and 1st Lt. Sam Allen, "State of Space Cyber: Midterm Report," Air Force Research Laboratory, Space Vehicles Directorate, Technical Rep. Version 1.0.8 (2014).

[37] A. Zeliff, M.J. Highbee, C.A. Price, 1st Lt John Guthrie, and C. Roman, "State of Space-Cyber: Final Report," Air Force Research Laboratory, Space Vehicles Directorate, Technical Rep. Version 1.0.6 (2015).

[38] D. Livingstone and P. Lewis, "Space, the final frontier for cybersecurity?" Chatham House; The Royal Institute of International Affairs, Technical rep. (2016).

[39] R. Murillo, R. Lindell, and P. Streander, "Interface Standard for CubeSat COMSEC Devices," The Aerospace Corporation, Technical rep. (2011).

[40] D. Fischer, I. Aguilar-Sanchez, B. Saba, G. Moury, C. Biggerstaff, B. Bailey, H. Weiss, M. Pilgram, and D. Richter, "Finalizing the CCSDS Space-Data Link Layer Security Protocol: Setup and Execution of the Interoperability Testing," in *AIAA SPACE Conference and Exhibition* (2015).

[41] B.J. LaMeres, S. Harkness, M. Handley, P. Moholt, C. Julien, T. Kaiser, D. Klumpar, K. Mashburn, L. Springer, and G.A. Crum, "RadSat – Radiation Tolerant SmallSat Computer System," in *29th Annual AIAA/USU Conference on Small Satellites* (2015).

[42] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, et al., "Meltdown: Reading kernel memory from user space," in *27th {USENIX} Security Symposium ({USENIX} Security 18)* (2018), pp. 973–990.

[43] Y. Kim, R. Daly, J. Kim, C. Fallin, J.H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *ACM SIGARCH Computer Architecture News*, IEEE Press (2014), vol. 42, pp. 361–372.

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

[45] "seL4-US," https://github.com/seL4-US, accessed December 2018.

[46] "The NASA Operational Simulator for Small Satellites," http://www.nos3.org, accessed December 2018.