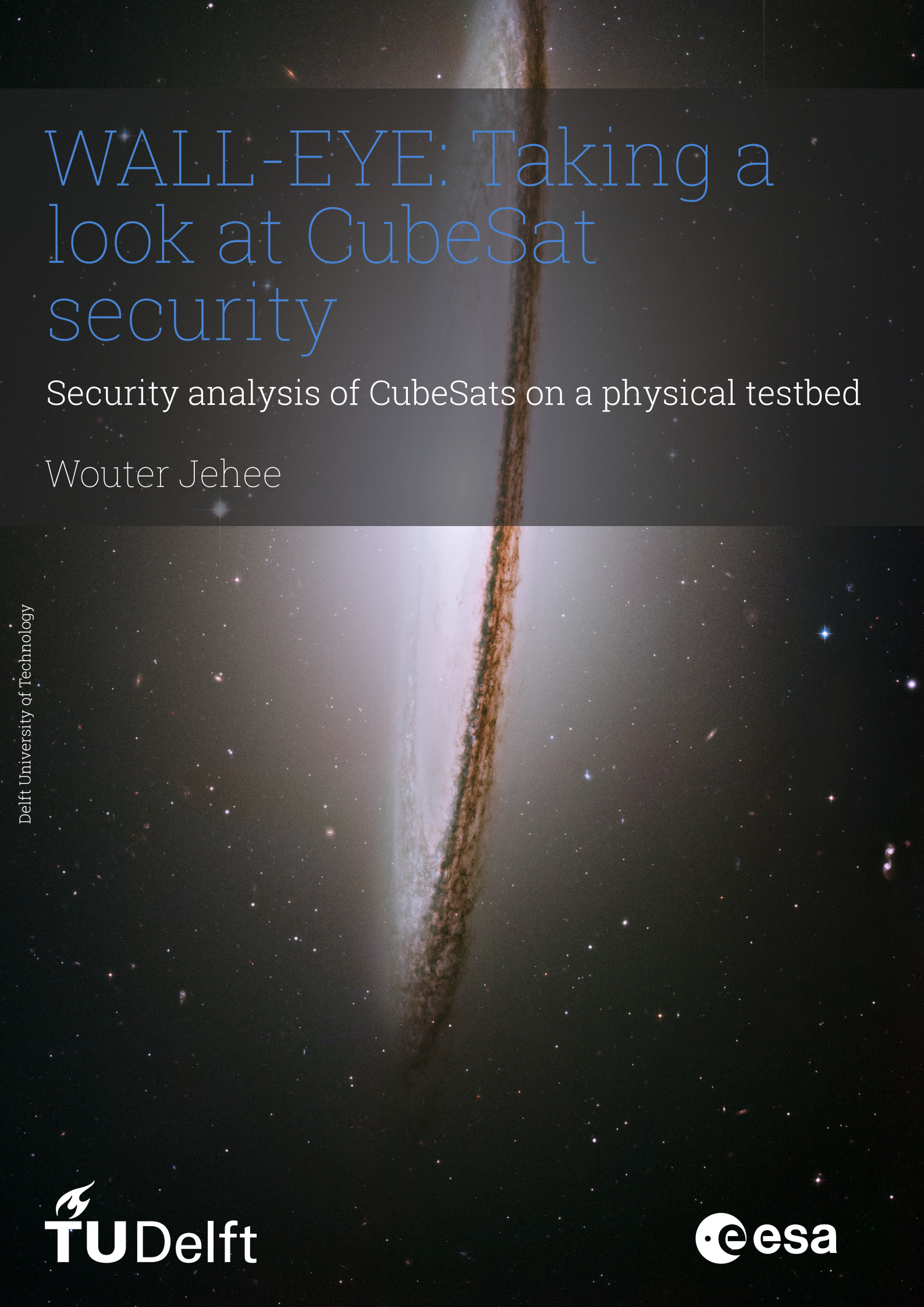


WALL-EYE: Taking a look at CubeSat security

Security analysis of CubeSats on a physical testbed

Wouter Jehee

Delft University of Technology

TU Delft

esa

WALL-EYE: Taking a look at CubeSat security

Security analysis of CubeSats on a physical testbed

by

Wouter Jehee

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Friday August 23, 2024 at 13:00

Student number:	4953355	
Project duration:	December 6, 2023 – August 23, 2024	
Thesis committee:	Prof. G. Smaragdakis,	TU Delft, Thesis advisor
	A. Voulimeneas,	TU Delft, Daily supervisor
	G. Iosifidis,	TU Delft
	Y. Roiron,	European Space Agency
	A. Atlasis,	European Space Agency

Cover:	Hubble mosaic of the majestic Sombrero Galaxy by ESA/Hubble under CC BY-NC 4.0
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft esa

Abstract

With the rise of new space, space missions are becoming increasingly more accessible. This is caused by the increased use of commercial-off-the-shelf components as well as the possibility of having multiple parties operating on a single satellite platform.

This development combined with a new attitude towards the security of these systems has exposed some flaws in current designs. These concerns are in the lack of defense-in-depth measures on board the satellite and the fully trusted nature of the internal bus.

In this thesis we perform a high-level risk analysis for CubeSat missions and map them to the SPACE-SHIELD framework. We then implement several mitigations based on the identified risks, with a focus on less explored research areas. The performance of the mitigations is then evaluated in order to test their viability for use in the industry.

We provide a simulator setup for testing and evaluating the mitigations. In order to improve the security of CubeSats we ran several fuzzing campaigns, which have led to the discovery of potentially vulnerable sections of code. Furthermore, we implemented mitigations to achieve network segmentation and end-to-end security on the internal bus of the device. The authenticated encryption scheme implemented for end-to-end security uses the NIST standard for lightweight cryptography known as Ascon. The measurements show that our reference implementation of this AEAD scheme has an overhead of 15% for message payload sizes of 200 bytes. Lastly, we contribute to several entries in the SPACE-SHIELD framework which were lacking before.

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
ADCS	Attitude Determination and Control System
AFL	American Fuzzy Lop
OBC	On Board Computer
CAN	Controller Area Network
CCSDS	Consultative Committee for Space Data Systems
C-I-A	Confidentiality, Integrity, Availability
COTS	Commercial-off-the-shelf
CPS	Cyber Physical System
CSP	Cubesat Space Protocol
GS	Ground Segment
IT	Information Technology
SDLS	Space Data Link Security (protocol)
SPP	Space Packet Protocol
SS	Space Segment
TC	Telecommand
TM	Telemetry
TT&C	Telemetry, Tracking and Command

1 Introduction

Deploying to space is becoming more accessible than ever, with the shift to using Commercial-off-the-shelf (COTS) products instead of custom components, the greater prevalence of open-source software in these components and the decreased cost of launching satellites into space. This has led to a significant increase in the number of devices launched into- and operating in space [1]. This trend change in the industry, referred to as "New Space" [2], has made security research into the space industry much more accessible in contract to the past [3]

Devices in space are Cyber Physical Systems (CPS), which have historically been considered inherently secure due to general their lack of connectivity [4], leaving security as an afterthought. With the new developments and transition to New Space, potential vulnerabilities have been identified in older systems [4]. The shift towards New Space brings along a different attitude towards security, namely defense-in-depth [2]. The principle of defense-in-depth is that security must be considered at multiple different layers, such that if one part of the system fails, other measures still protect the system. For the older systems, there has mostly been a single point of failure, which is the security of the link between the spacecraft and the ground station.

Another more recent development are hosted payloads, where third-parties can bring custom hardware on board of another satellite. Examples of hosted payloads include the European Data Relay System hosted on a commercial telecom satellites and a classified US payload on a Norwegian satellite. Satellite-as-a-Service (SaaS) models, which allow third-parties to run their own experiments on the platform of other satellites. This is very financially attractive [5] since not everyone needs to own their own satellite. This approach also brings along it's fair share of security risks [6], as there is a lack of understanding on the security of payloads and the internal connection to other components [7]. In order to verify the security of the hosted code, extensive testing needs to be done, unless another approach is used.

A zero trust architecture on board the satellite accounts for the fact that hosted payloads and/or programs might be malicious by not trusting them from the start. Such a model reduces the need for an extensive security analysis on the components of the third-parties involved, but it does require a significant change in the design. Research on internal security has been done in automotive [8] [9], but not for the space sector.

A large portion of new space missions use smaller satellites such as CubeSats. CubeSats might be small, but they have important roles in crucial projects such as communication infrastructure [10] [11] and reconnaissance [12]. Therefore, security is critical for CubeSats and must be considered when designing a system [13]. Although there have been security analyses on CubeSats, none have done an applied security analysis on real hardware.

In the past, standardization has been limited with regards to the security of space systems [14]. Several initiatives in the industry are trying to change this. One such example is the Space Attacks and Countermeasures Engineering Shield (SPACE-SHIELD) [15], a knowledge base for cyber threats and mitigations for space systems. The SPACE-SHIELD framework is still relatively new however, so many techniques are not yet documented in detail or at all.

Due to the previous factors, this research aims to answer the following main research question, by answering the 4 defined subquestions.

How to establish a methodology to manage the security of CubeSats?

- What attack vectors are CubeSats vulnerable to?
- How can we create an environment for doing applied security testing?
- Which SPACE-SHIELD mitigations can we apply to improve security?
- How can we improve the SPACE-SHIELD framework?

The rest of this thesis is structured as follows. Chapter 2 will provide background information on the space industry, CubeSats specifically and general security practices. Chapter 3 contains the risk analysis and how it relates to the SPACE-SHIELD framework. Chapter 4 describes details about the physical and simulated testing environment. Chapter 5 explains the mitigations and their implementation. Chapter 6 shows the results of the security testing and includes a performance analysis of one of the implemented mitigations. Chapter 7 discusses the results and limitations of the study. Chapter 8 concludes this thesis and suggests topics for future research.

2

Background

This section provides background information about some of the core technologies used for the space sector and for CubeSats specifically. It also discusses the security of these technologies as well as any related work.

2.1. Space environment

Operations in space usually consist of three main elements, as can be seen in Figure 2.1. The Space Segment (SS) includes all of the assets in space, it listens for TeleCommands (TC) and sends out TeleMetry (TM). The Ground Segment (GS) is a station on the ground that sends TC and receives TM. The user segment is the third segment, it provides some kind of service and is therefore optional, since not all satellite missions provide a service directly to users (such as Earth observation missions). The last important thing to consider is the communication between these segments (the TC and TM), which we call the link segment, even though it is not an actual (set of) device(s). Communication over the link segment is usually done over Radio Frequency (RF), although there have been developments in using optical communication for Inter-Satellite Links (OISL) [16].

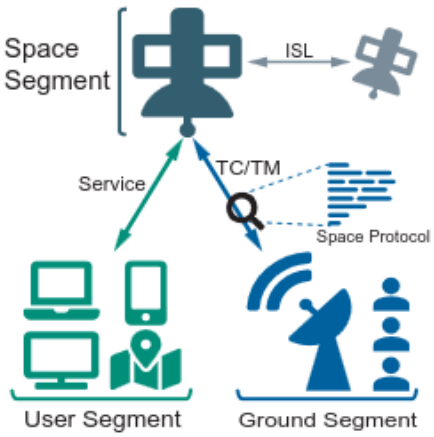


Figure 2.1: Overview of different segments in satellite operations [3].

Architecture

A Reference Architecture (RA) can be used to provide a better understanding of complex systems. RAs can be used at various stages of development, e.g. to define the requirements of a system but also to identify weaknesses in the design. For a cybersecurity analysis, the functional and communication viewpoints are the most critical [17]. The functional viewpoint defines what components perform which tasks, whereas the communication viewpoint describes how the components inter-operate.

One of the outputs of the Space AVionics Open Interface aRchitecture is an example of such a reference architecture for space systems. The SAVOIR initiative aims to: "federate the space avionics community and to work together in order to improve the way that the European Space community builds avionics sub-systems" [18]. The avionics reference architecture, as described by SAVOIR, can be seen in Figure 2.2.

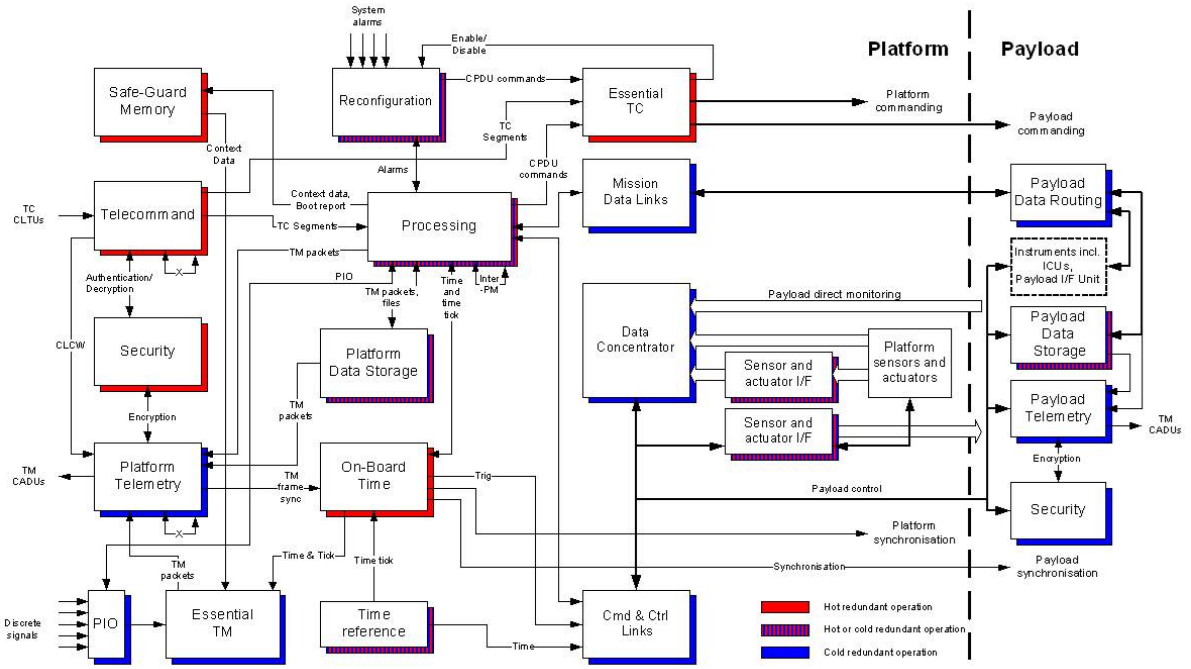


Figure 2.2: SAVOIR functional avionics reference architecture [18]

This reference architecture specifies which functional components are necessary in most space systems and how these components interact with each other, thus combining the functional and communication viewpoints. Physical components in the final design may fulfill the tasks of multiple functional components.

Spacecraft operate in a very hostile environment, this is not only a problem in and of itself, but it also makes them very hard to service in case something does go wrong. In the diagram one can see that some operations have redundancy built in, this allows the spacecraft to continue operating that component in case of failure. As such most spacecraft are at least one failure tolerant.

We generally refer to the main components required for the operation of the satellite as the platform. The payload modules are mission specific components such as sensors, this distinction can also been in Figure 2.2. The payload modules make use of the power and communication of the satellite, but operate independently from it [5]. Such payloads can be owned by the developer of the satellite, but also by third-parties. Bringing along a module on board of an existing platform is known as hosting. This way the third party does not have to have it's own satellite, but can make use of an already existing platform. Payload computers on board newer satellites take this even further, by operating on a multi-tenant model, which allows multiple third parties to access a single payload computer and it's mission specific equipment. In the multi-tenant model, a single payload supports multiple entities, either through virtualization or some other kind of memory isolation [6].

Computers on board of satellites have a wide range of capabilities, some devices are very constrained whilst others support fully fledged desktop operating systems. We classify the software running on these (embedded) systems into 3 classes [19]:

- Type 1: General purpose operating systems (e.g. Linux)
- Type 2: Embedded / real-time operating systems (e.g. FreeRTOS)
- Type 3: Monolithic / bare-metal systems

Protocols

In the space sector, several protocols are defined by the CCSDS to be used for the link between the ground- and space segment. An overview of the different recommended space communication protocols can be seen in Figure 2.3.

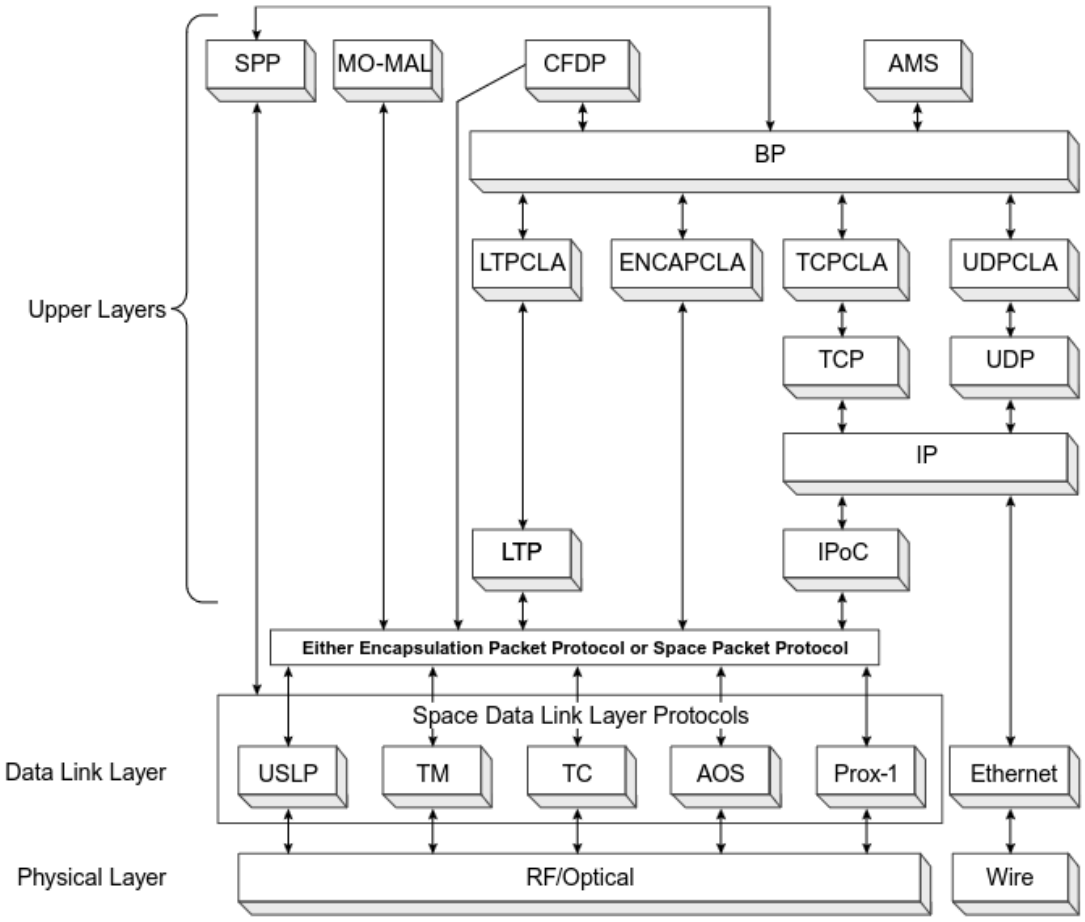


Figure 2.3: Overview of space protocols [20]

For the security of the link, the SDLS standard [21] and it's extended procedures are used [22]. An overview of how security is applied to the protocols can be seen in Figure 2.4.

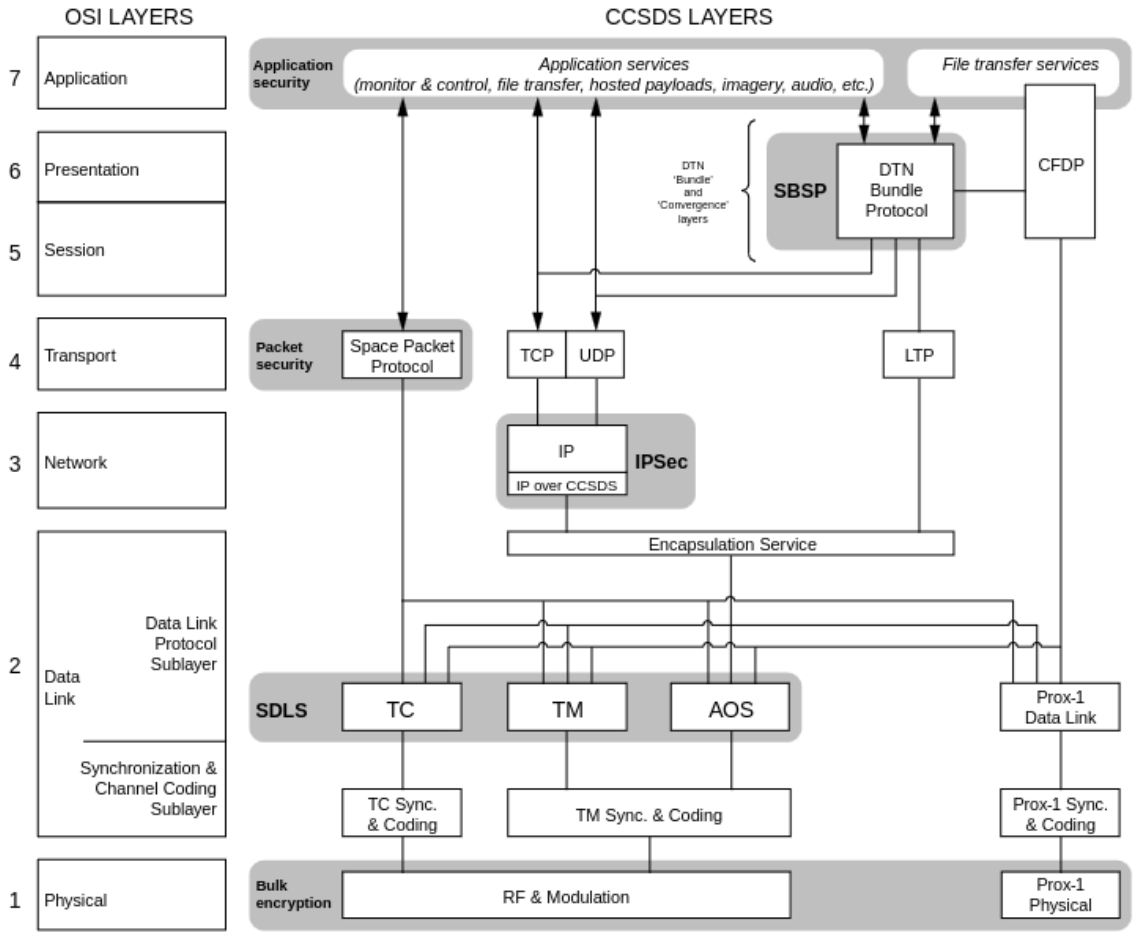


Figure 2.4: Security of CCSDS protocols [23]

Once on the satellite, information also has to flow to the right place inside of the satellite, since the final design of a satellite often has multiple physical components that together achieve all the required functionality. These components need to be able to communicate with each other in order to operate. For this internal communication, satellites can use various different technologies, such as i^2c, CAN, UART, MIL-STD-1553, Time-Triggered Ethernet (TTE), etc. Many satellites use a single bus to connect all the components on board, as this is often the simplest and cheapest solution.

One such bus is CAN, it is used in both the automotive industry [24] as well as in space [25]. CAN is a simple broadcast-based, message-oriented protocol meant for efficient communication. Broadcast-based protocols are of particular interest when considering security, since there practically is none, any component can listen in on all communication and can send arbitrary messages [4].

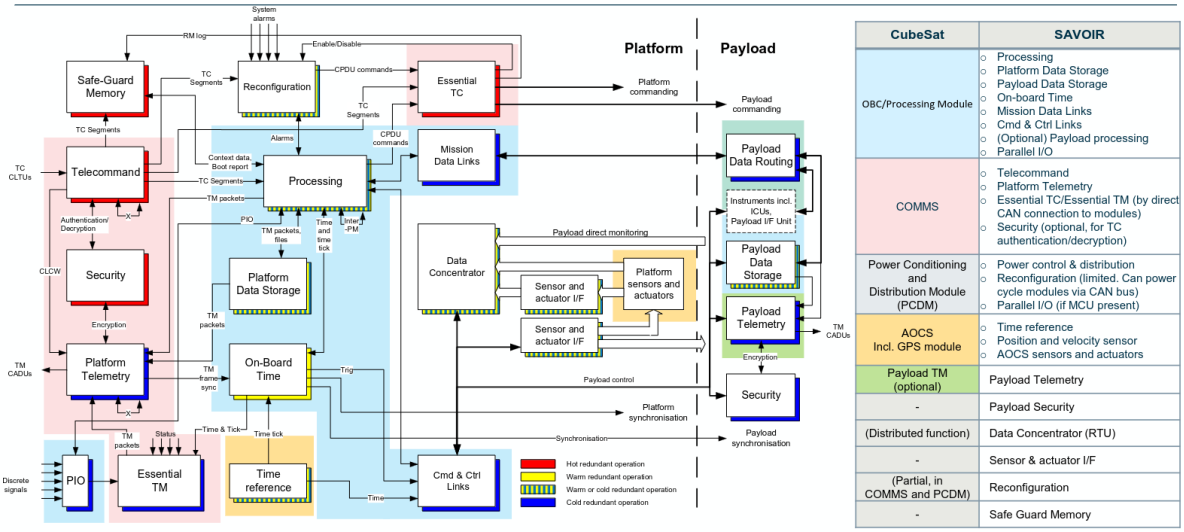
It is also possible to connect internal components using point-to-point based protocols, such as SpaceWire [26] and it's successor SpaceFibre [27]. This approach might mitigate some of the security concerns with broadcast based protocols, but has it's own drawbacks.

2.2. CubeSats

CubeSats [2] are part of a class of satellites called nanosatellites. Nanosatellites come in sizes ranging from 1U to 12U (1U or unit, is $10 \times 10 \times 10$ cm) and are currently the most prevalent class of satellites [3]. They are often used by universities and research institutes because of their relatively low cost.

Their small size obviously limits the amount and size and size of components that can be integrated, this in turn also limits their computation power. Many CubeSats use a CAN bus for communication between internal components.

A diagram showing SAVOIR applied to cubesats [28] can be seen in Figure 2.5. Each color in the diagram on the left side is mapped to a component in the table on the right, the functional components within this color are fulfilled by the component on the right, the names of these components are also listed in the table.



CubeSat	SAVOIR
OBC/Processing Module	○ Processing ○ Platform Data Storage ○ Payload Data Storage ○ On-board Time ○ Mission Data Links ○ Cmd & Ctrl Links ○ (Optional) Payload processing ○ Parallel I/O
COMMS	○ Telecommand ○ Platform Telemetry ○ Essential TC/Essential TM (by direct CAN connection to modules) ○ Security (optional, for TC authentication/decryption)
Power Conditioning and Distribution Module (PCDM)	○ Power control & distribution ○ Reconfiguration (limited. Can power cycle modules via CAN bus) ○ Parallel I/O (if MCU present)
AOCS Incl. GPS module	○ Time reference ○ Position and velocity sensor ○ AOCS sensors and actuators
Payload TM (optional)	Payload Telemetry
-	Payload Security
(Distributed function)	Data Concentrator (RTU)
-	Sensor & actuator I/F
(Partial, in COMMS and PCDM)	Reconfiguration
-	Safe Guard Memory

Figure 2.5: SAVOIR functional reference architecture for CubeSats [28]

CubeSats most commonly use either the Space Packet Protocol (SPP) or Cubesat Space Protocol (CSP) [10]. Since CubeSats have limited resources, a lightweight communication protocol such as CSP [29] is preferred. The goal of CSP is to provide developers with a similar protocol to TCP/IP, but with less overhead. The CSP stack also provides some high-level functionality for debugging, allowing users to ping nodes, peek into memory, etc. CSP provides optional encryption and authentication implementations using XTEA and HMAC-SHA1 respectively, which are both known to have cryptographic weaknesses [2]. XTEA has been removed in the newest versions of CSP, but since devices using this technology are not trivial to update, there might still be devices in operation using these features.

CubeSats generally use type 2 software such as FreeRTOS [30] or ZephyrOS [31], but bare-metal systems are also used. There is also a more recent development to use more type 1 software, such as fully-fledged Linux operating systems [32], which consume more resources but are also more flexible.

2.3. Related work

There have been various studies into the assessment and improvement of security for space systems and CubeSats, we highlight some of those studies here.

In order to assess vulnerabilities, some have made attack tree analyses for attacks on CubeSats [13], showing different attack paths based on the attackers goals. There also exists a framework for triaging vulnerabilities for space similar to MITRE ATT&CK, known as SPACE-SHIELD [15].

Fuzzing is a technique for testing software that involves continuously sending randomly generated inputs to a program in order to test a program [33]. This technique is becoming increasingly popular to test the security of various devices and protocols. Besides fuzzing there is also symbolic execution, which looks at which parts of the program are being executed based on the input in order find new paths more quickly [34]. Researchers have used fuzzing to try and find vulnerabilities in firmware by simulating the firmware through a process known as rehosting [35]. Other work specifically fuzzed the firmware of CubeSats using this technique [36]. In order to reduce vulnerabilities, some have tried porting Rust to space systems, they also fuzzed CSP [37]

When hosting payloads, it can be beneficial to do sandboxing in order to limit the capabilities of the payload. This can be done through Virtual Machines (VMs), hypervisors, containers or application sandboxes [38]. Another method to improve the security of CubeSats is through Trusted Execution Environments (TEEs), which was looked into by [39].

In automotive there has been research into several methods of securing the internal communication bus between components in the vehicle:

- Secure communication on CAN bus [40] [8]
- Access control for vehicles [9]
- Several countermeasure similar to an IDS/IPS are suggested in [41]
- Anomaly detection based on packet timing [42]

Since CubeSats have relatively limited resources, some have looked into lightweight cryptography for communication [43] [44].

3

Methodology

This section discusses the methodology used to perform the security analysis. In order to identify the most vulnerable parts of the system, we analyzed the risks associated with a CubeSat space mission, following the risk analysis framework of the European Space Agency (ESA).

3.1. Overview

This risk analysis focuses on a generic CubeSat architecture, classifying the risks that are possible on such as a system. Any mission specific risks are considered out of scope, such as the threats to the Ground Segment (GS) and the supply chain procedures that happen before the launch. The link between the Space Segment (SS) and GS is taken into account however.

We consider the following four factors for security:

- Confidentiality (C): The importance of this factor is very much dependent on the nature of the mission.
- Integrity (I): The most critical for maintaining control over the satellite, also affects reputation if compromised.
- Availability (A): Not the most important factor to consider as the device can run autonomously for a limited time.
- Non-repudiation: Importance depends on the mission and the entities it will communicate with. The specifics of the mission must define how this is handled. This factor is thus not studied in depth in this thesis.

The following are the objectives we consider in the risk analysis:

- General objectives:
  - Protect ESA member states investments in space
  - Protect ESA & member states reputation, image & interests
  - Provide secure space systems, developed in a secure environment
- Mission specific objectives:
  - Ensure confidentiality of the mission data (limited time) and status of the space segment
  - Preserve integrity of the mission data
  - Maintain control over the on board components (OBC, payload, etc.)

Metrics

For us to prioritize the risks, we need metrics for how likely something is to happen and how severe the consequences are if they do. These metrics are the *likelihood* and *severity*, which are defined below.

- Rare: < 1% probability
- Unlikely: 1-5% probability
- Possible: 5-10% probability
- Likely: 10-50% probability
- Very likely: > 50% probability

In Table 3.1, we define the impact of the various severity levels on the technology and security of the system. The technology of the system refers to how well the device is able carry out it's mission.

Severity	Technology	Security
Low	The impact is minimal, with little to no effect on the exploration, science, or technological objectives. This severity level suggests that the risk is so minor that it could almost be ignored.	The flaw, even if exploited, would have an insignificant impact on the system. The security breach would be minor and easily contained, with no substantial effect on the mission.
Low/medium	The impact would cause minor degradation of the exploration or science value, affecting specific, non-critical parts of the project. The severity here suggests that while there is an impact, it is limited and does not threaten the overall success of the mission.	The flaw, if exploited, would have a limited impact on the system, causing minor security issues that are unlikely to significantly affect the overall mission. The damage would be contained and manageable.
Medium	The impact would cause a noticeable degradation of the exploration or science value on a few instruments or technological aspects. While not crippling, the severity is significant enough to cause concern and warrant focused mitigation efforts.	The flaw, if exploited, would result in moderate damage to the system, affecting its functionality or security in specific areas. The impact is concerning but manageable.
Medium/high	The impact would lead to a significant degradation of the science or technology value across most instruments or units. The severity here indicates that while the project could continue, its objectives would be severely compromised.	The flaw, if exploited, would cause major damage, leading to a significant security breach that could severely affect the system's integrity. This impact is considerable and could jeopardize large parts of the mission.
High	The impact would be catastrophic, causing a failure to achieve the exploration, science, or technology objectives. This severity level represents the maximum possible harm, where the risk could critically undermine the entire project or mission if it materializes.	The flaw, if exploited, would cause severe and irreparable damage significantly compromising the entire system or mission. This level of severity means that the impact on security would be devastating, even if the flaw could be corrected.

Table 3.1: Severity levels for different factors

In Table 3.2, we classify the risk levels based on the likelihood and severity of an incident occuring.

Likelyhood / Severity	Very low	Low	Medium	High	Very High
Rare	Very low	Very low	Very low	Low	Low
Unlikely	Very low	Very low	Low	Low	Medium
Possible	Very low	Low	Low	Medium	High
Likely	Low	Low	Medium	High	Very high
Very likely	Low	Medium	High	Very high	Very high

Table 3.2: Risk level based on likelihood and severity

Lastly, we consider the following possible treatments for each risk:

- Mitigate: reduce likelihood and/or impact of the risk until acceptable
- Avoid: eliminate the cause of the risk, by either stopping the activity all together or a fundamental change in the activity that completely avoids this risk
- Transfer: transfer the risk to a third party
- Accept: accept the consequences of a risk if they happen, do not provide any additional mitigations
- Out-of-scope: out-of-scope for this research, not (fully) analyzed

3.2. Threat sources

Threat sources can be grouped into multiple categories [45], namely adversarial-, insider-, environmental- and structural sources. Given the scope of this research is focused on the security domain and not safety, we will only be considering adversarial and insider sources, these will be referred to as threat actors from now on.

Threat actors can again be grouped according to multiple criteria. The four major types of threat actors described by [46] are:

- States seeking strategic advantage
- "Organized criminal efforts for financial gain"
- "Terrorist groups seeking recognition"
- "Individual hackers proving their skill"

These threat actors are a good start, but only adversarial sources and not insider sources are included, we therefore add insider threats as well. Threat actors can also be subdivided into three groups, those being: individuals, groups and organizations [17], we include this categorization in our list of threat actors. The threat model used in this case is based on the ESA standard for space missions, the threat actors we consider can be seen in Table 3.3. In the case we divide the threat actors between inside and outside threats (insider and adversarial sources).

Threat actor	Type	Internal/External	Objective
Public	Group	External	Defeat
Hacker/script kiddie	Individual	External	Defeat
Disgruntled employee	Individual	Internal	Resist
Hacktivist/hacking group	Group	External	Resist
Insider helping other	Group	Internal	Deter
Foreign espionage	Organization	External	Deter
Unfunded terrorist	Individual	External	Deter
State sponsored	Group	External	Deter
Nation state	Organization	External	Deter → Accept

Table 3.3: Overview of threat actors considered

3.3. Risk register

The risk register shows the identified risks and how they fit into the analysis, Table 3.4 shows an overview with the area (broad category), relevant threat actors and assets, as well as the severity, likelihood and treatment for that risk. The subsections after are divided per area and provide a mapping to the SPACE-SHIELD [15] framework as well as a more detailed explanation for each of the risks.

Risk ID	Area	Threat Actors	Assets	CIA	Severity	Likelihood	Treatment
1	Design & Development	Cyber criminal Insider threat	Design	CI	Medium/High	Possible	Out-of-scope
2	Ground Segment	Cyber criminal Hacktivist group Insider Threat	Ground Segment	CIA	Medium/High	Likely	Out-of-scope
3	Supply Chain, Testing & Validation	Cyber Criminal Insider Threat	Design Space Segment	I	Medium/High	Rare	Out-of-scope
4	On Board Processing & Storage	Cyber Criminal Insider Threat	Mission Data Space Segment	IA	Medium	Rare	Mitigate
5	On Board Processing & Storage	Cyber Criminal Insider Threat	Mission Data Space Segment	IA	Medium	Rare	Mitigate
6	On Board Processing & Storage	Cyber Criminal Insider Threat	Mission Data Space Segment	CIA	Medium	Rare	Mitigate
7	TT&C, Payload communication	Cyber criminal Hacktivist group Insider Threat	Space Segment	I	Medium	Rare	Mitigate
8	TT&C, Payload communication	Cyber criminal Hacktivist group Insider Threat	Telecommands (TC) Telemetry (TM)	C	Low	Unlikely	Mitigate
9	TT&C, Payload communication	Cyber criminal Hacktivist group Insider Threat	Space Segment	A	Low/Medium	Unlikely	Accept

Table 3.4: Risk register

Design & Development (risk 1)

Reconnaissance	Resource development	Defense evasion	Discovery
Gather Victim Mission Information Gather Victim Org Information Phishing for Information	Develop/obtain Capabilities	Impair Defences	Key Management Policy Spacecraft Components System Services Trust Relationships

Table 3.5: SPACE-SHIELD mappings for risk 1

In design and development, confidentiality is affected when there is leakage of information about the design specification. This can occur by an insider threat leaking the information (on accident or on purpose), or by exfiltration by a cybercriminal. The leaked information can include information about the mission and the organization, but also specifics about the hardware and protocols being used. Based on the specifics of the design, a threat actor can also develop capabilities to attack the system at another time.

Integrity of the design can be affected if the threat actor alters the design beyond the original intention. This falls under the develop/obtain capabilities technique, as well as the impair defenses technique, since faults can be introduced that can be exploited later.

Risks related to the design and development are considered out of scope for this thesis because this is specific to the project and not a general CubeSat mission.

Ground segment (risk 2)

Reconnaissance	Resource development	Initial access
Gather victim mission information Gather victim org information Phishing for information	Compromise infrastructure Compromise account	Ground segment compromise Valid credentials

Table 3.6: SPACE-SHIELD mappings for risk 2

Compromising the ground segment can have quite severe consequences. The attacker can use it to get initial access on a satellite, but also steal credentials for later use. Any compromised infrastructure and accounts can also be used for further exploitation, affecting the integrity of the system. Besides stealing credentials, the attacker is also able to retrieve information about the mission and organization, affecting confidentiality. Lastly, a cyber attack on the ground segment (such as a ransomware attack) can also cause a loss of availability, as the systems can no longer be operated.

We do not look into risks for the ground segment in more detail because the ground segment is more similar to a regular IT system, and thus not the main object of this study.

Supply chain, testing & validation (risk 3)

Initial Access	Persistence	Impact
Supply Chain Compromise	Backdoor installation	Data Manipulation Permanent / temporary loss to Telecommand Satellite Resource damage Resource Hijacking Saturation/exhaustion of Spacecraft Resources Service stop

Table 3.7: SPACE-SHIELD mappings for risk 3

This risk includes unintended or malicious alterations made to the device in the pre-launch phase [47] (during assembly, transport, etc.), which can result in failure or vulnerabilities at a later stage of the integration process. These modifications directly affect the integrity of the system. This can include modification of software (on board software modification), installing malicious hardware components (hardware trojans), etc.

In this study we do not investigate these risks as they are project specific.

On board processing & storage

Risk 4

Execution	Credential Access	Discovery
Modification of on board control procedures Native API Payload exploitation to execute commands	Adversary in the middle	Spacecraft components System services
Lateral Movement	**Collection**	**Impact**
Lateral movement via common avionics bus Compromise the satellite platform starting from a compromised Payload Compromise a payload Starting from the main satellite platform	Adversary in the middle	Temporary loss to telecommand satellite Saturation/exhaustion of spacecraft resources Resource hijacking Service stop

Table 3.8: SPACE-SHIELD mappings for risk 4

This risk refers to corruption of an OBC or payload through a malicious software update or patch, known as a post launch supply-chain attack [47]. This allows an adversary to install malicious software on one of the on board computers, affecting the integrity of the system. It can impact availability as well, as the original software can be overwritten, potentially locking out authorized parties.

This risk is considered rare as it requires a high level of understanding of the hardware target and update procedures. It also requires that the attacker has either compromised the link segment or already has a foothold in the space segment.

Risk 5

Execution	Credential Access	Discovery
Modification of on board control procedures Native API Payload exploitation to execute commands	Adversary in the middle	Spacecraft components System services
Lateral Movement	**Collection**	**Impact**
Lateral movement via common avionics bus Compromise the satellite platform starting from a compromised Payload Compromise a payload Starting from the main satellite platform	Adversary in the middle	Temporary loss to telecommand satellite Saturation/exhaustion of spacecraft resources Resource hijacking Service stop

Table 3.9: SPACE-SHIELD mappings for risk 5

This risk refers to flaws in the OBC (software) allowing attackers to execute arbitrary code. This leads to loss of integrity as the original software is overwritten. Availability is also easy to affect if arbitrary code execution is possible. The flaw could also be limited to only crashing the OBC, which would only cause a loss of availability.

On board data handling (risk 6)

Execution	Credential Access	Discovery
Modification of on board control procedures Native API Payload exploitation to execute commands	Adversary in the middle	Spacecraft components System services
Lateral Movement	**Collection**	**Impact**
Lateral movement via common avionics bus Compromise the satellite platform starting from a compromised Payload Compromise a payload Starting from the main satellite platform	Adversary in the middle	Temporary loss to telecommand satellite Saturation/exhaustion of spacecraft resources Data manipulation

Table 3.10: SPACE-SHIELD mappings for risk 6

This risk refers to a malicious hosted payload or third-party software module affecting the operation of the device. The adversary can trivially flood the internal bus with data, exhausting the satellites resources as well as blocking communication between other components [48]. Loading malicious software is not a breach of integrity by itself, since the third-party was already allowed to load custom software / hardware on board. However, the adversary has a direct foothold in the system and can use the internal bus to talk to the other components

The bus has a broadcast structure, meaning any component can spy on all messages exchanged on the bus [40], affecting confidentiality if the attacker is able to communicate back. The adversary can also send malicious bus messages to command critical systems [49]. If mitigations are in place to prevent direct commands, flaws in the on board data handling could still allow the attacker to move laterally.

TT&C, payload communication

For risks in this category, we consider threat actors listening in on and tampering with the link segment. As discussed earlier, communication is almost always done over RF, so the threat actors must have the appropriate equipment.

Risk 7

Reconnaissance	Initial Access	Credential Access
Active scanning (RF/Optical)	Direct attack to space communication links Trusted relationship Valid credentials	Brute force
Command and control	**Impact**	**X**
Protocol tunnelling Telecommand a spacecraft TT&C over ISL	Loss of spacecraft telecommanding Permanent loss to telecommand satellite Resource hijacking	X

Table 3.11: SPACE-SHIELD mappings for risk 7

In this risk scenario, an unauthorized party is able to control the during it's operational phase or during the Launch & Early Orbit Phase (LEOP). This can be done through a vulnerability in the system or by using valid credentials. These credentials could be stolen during an earlier attack, brute forced or a trusted relationship with another device.

These risks result in either a full compromise of the system (i.e. hijack attack) or the ability for the threat actor to telecommand the satellite, compromising the integrity of the system.

Risk 8

Reconnaissance	Credential Access	Discovery	Collection
Passive interception (RF/optical)	Communication Link Sniffing	Key management policy discovery Spacecraft's components discovery System service discovery	Data from link eavesdropping

Table 3.12: SPACE-SHIELD mappings for risk 8

This risk considers the risk of an unauthorized party receiving and decoding telemetry/telecommand data. This affects confidentiality as the adversary is able to read incoming and outgoing data. Telemetry data is generally considered more impactful for confidentiality, but that also depends on the mission. Telecommand data is not particularly interesting in and of itself, but it is relevant for reconnaissance purposes. This kind of data can give insight into which services and components are available on the device, which can be relevant for further attacks and techniques.

Risk 9

Impact
Ground segment jamming Spacecraft jamming Saturation of inter satellite links Transmitted data manipulation

Table 3.13: SPACE-SHIELD mappings for risk 9

This risk is about denial of service attacks on the communication with the spacecraft. By overwhelming the link segment, the device can not send TM or receive TC commands, this affects the availability of the device. The spacecraft can operate autonomously for some period of time, which decreases the impact of such loss of availability.

4

Experimental setup

This section describes the physical testbed as well as the simulated setup we used to perform all the tests. The simulated setup was used to perform the initial testing for each of the mitigations. Whereas the physical test bed was used to verify the mitigations.

4.1. Physical test bed

For this research, access was granted to a lab version of a CubeSat, known as a Flatsat, a picture of the Flatsat can be seen in Figure 4.1. The Flatsat is basically a folded out version of the normal compact form. This allows for easy access to all the components for testing purposes.

This particular Flatsat (Flatsat-P3) has 12 slots for components, which are divided into two halves with 6 components each. As can be seen in Figure 4.1, only one half of the FlatSat is in use, the most relevant installed components are:

- 2 OBC-P3 modules
- ADCS computer (Cortex M7)
- Power Conditioning and Distribution Unit (PCDU)
- Reaction wheel

In Figure 4.2 one can see how these components are connected.

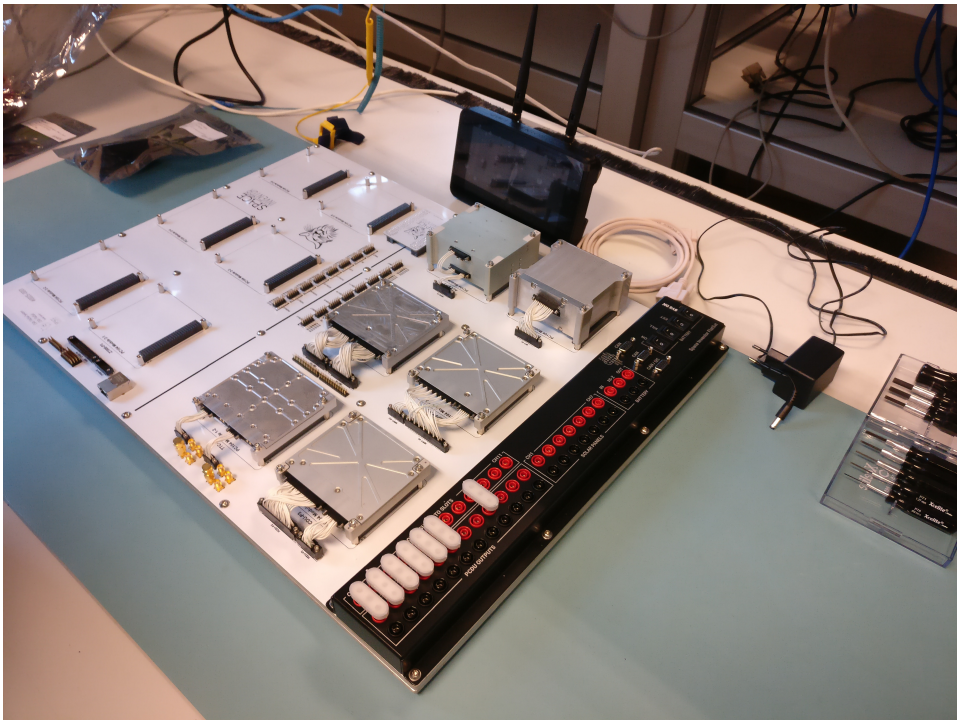


Figure 4.1: Picture of the test bed

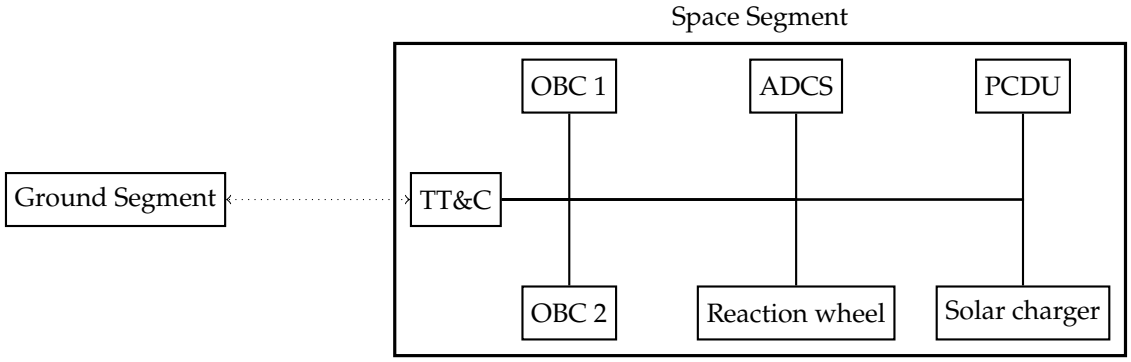


Figure 4.2: Diagram showing the components of the physical testbed and how they are connected

The specifications of the on board computers (OBC-P3) on the testbed are as follows:

- ARM Cortex-M7
- 384 kB SRAM
- 64 GB eMMC storage
- 32 kB FRAM memory
- 2 MB flash memory

The firmware for the OBC runs on hardware specifically designed for the space environment, it is an application built on top of FreeRTOS [30]. The communication stack used on the FlatSat is CSP [29], which transmits over CAN to the other components in the system. It also uses a library known as libparam [50], which is a library to manage settings on the system at runtime. For C standard library functionality, the setup uses picolibc [51]. An overview of the software and the versions used can be seen in Table A.1.

4.2. Simulating the lab setup

Even though the lab setup is meant for easier testing than on a normal CubeSat, there are still some limitations with testing. In order to test things, the firmware of the device needs to be flashed with the new code every time, this is a somewhat time-consuming process. Furthermore, it's quite hard to access logs and crash information on embedded devices. Lastly, in order to interact with the system we have to send messages over a physical medium, causing latency which in turn makes testing slower.

In order to do this, we had to remove some hardware specific code and change out some libraries that were used in the physical setup [52]. To simulate the OBC code, we needed to upgrade the FreeRTOS version to be able to simulate it on POSIX (and Windows / MAC) systems [30]. A list of all major software packages and their versions can be found in Table A.1 in Appendix A In order to enable networking capabilities on the host machine, we use the FreeRTOS extension FreeRTOS-Plus-TCP which in turn uses libslirp [53] for network access.

In Figure 4.3 one can see the setup for simulating the environment of the physical test bed, each OBC node is a FreeRTOS simulator. We only simulate on board computers in this setup, but any component could be simulated in this environment. The internal communication between the nodes is done via ZMQ with a central proxy which broadcasts all the messages to all nodes, just like CAN is used on the physical testbed (Figure 4.2). The external node represents any communication coming from the outside, this can be done through any medium implemented by the bridge node, the simplest protocol was UDP so this was used.

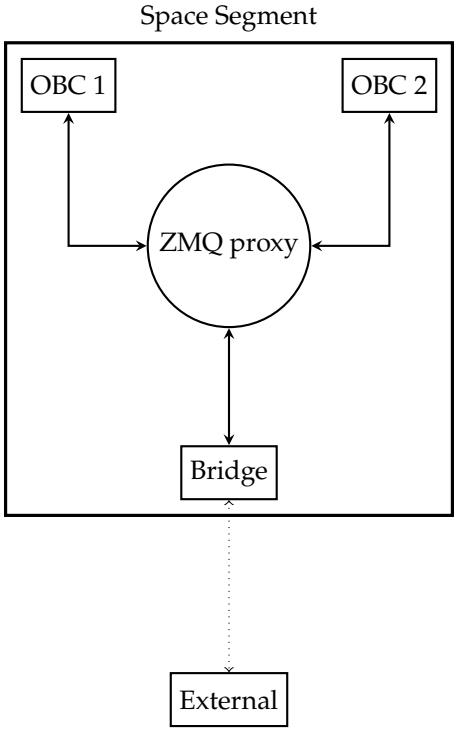


Figure 4.3: Diagram showing how the test bed is simulated

5

Mitigations

In this chapter, we consider different mitigations for the risks identified in the risk analysis and classify them as per the SPACE-SHIELD framework. The mitigations are divided into two sections. Section 5.1 introduces mitigations for during the development phase. Whilst Section 5.2 discusses mitigations that are used in the operational phase.

Because the mitigations related to link security are already considered state of the art by researchers, we wanted to focus on the identified risks not covered by the literature.

Table 5.1 shows a general overview of which tactics are relevant for each implemented mitigation, which techniques it aims mitigate, and which SPACE-SHIELD mitigations it covers. Lastly, it shows which risks identified in the previous section it helps to mitigate.

Mitigation	Tactics	Techniques	Mitigations	Phase	Risks
Fuzzing	Resource development	Develop/obtain capabilities	Fuzzing/testing	Development	5, 7
Security gateway	Lateral movement Collection Exfiltration	Lateral movement via common avionics bus Compromise the satellite platform starting from a compromised payload Compromise a payload starting from the main satellite platform Adversary in the middle Exfiltration over payload channel Exfiltration over TM channel Side-channel exfiltration	Network segmentation Filter network traffic Defense-in-depth Access control (application of least priviledge principle)	Operational	6, 9
Authenticated encryption	Discovery Lateral movement Collection Exfiltration	Spacecraft's Components Discovery System Service Discovery Lateral movement via common avionics bus Compromise the satellite platform starting from a compromised payload Compromise a payload starting from the main satellite platform Adversary in the middle Exfiltration over payload channel Exfiltration over TM channel Side-channel exfiltration	Authenticated encryption CCSDS SDLS sequence numbers On board authentication for executing critical commands Access control (application of least priviledge principle) Anti-replay protection Defense-in-depth	Operational	4, 6

Table 5.1: Implemented mitigations' mapping to SPACE-SHIELD tactics, techniques and mitigations

5.1. Mitigations during development

This section describes a mitigation that can be applied during the development stages of a mission. Mitigations at this step in the process help with identifying vulnerabilities before the operational phase.

5.1.1. Fuzzing

Fuzz testing (fuzzing) allows developers to detect flaws in the system during the development stage. In SPACE-SHIELD this would fall under the software vulnerabilities and space protocol vulnerabilities sub-techniques of develop/obtain capabilities. We run the fuzzer on two different versions of CSP and libparam in order to test both the version currently being used on the FlatSat, as well as a newer version.

We originally considered directly fuzzing the physical testbed, but there are some difficulties with this approach. The way the devices are setup makes it hard to retrieve logs and other information as soon as the device crashes. That is because the device has four flash slots which can contain programs; as soon as a program crashes, the device will load the next slot in line, overwriting the memory of the previous run. That is why we perform fuzzing on the simulated version of the OBC instead.

For fuzzing, we use AFL++ [54] as well as it's CMPLOG/Redqueen [34] extension. Since the fuzzing target is a networked program we need a way for AFL to send packets. Preeny [55] is library that can be preloaded to change socket operations such that STDIN and STDOUT can be used as network input and output. Preeny is the most realistic in terms of results, any crashes will crash the program again since they directly emulate being sent on the network.

When fuzzing, a vulnerability might be present but might not always cause a crash in the program. That is why sanitizers exists, they detect behavior that can lead to exploitation and crash the program. Several fuzzing sanitizers are listed below:

- Address sanitizer: detects use-after-free, buffer overflows, and other memory corruption vulnerabilities.
- Memory sanitizer: detects uninitialized memory accesses.
- Undefined behavoir sanitizer: detects undefined behavior as specified by C and C++ standards.
- Control flow integrity sanitizer: detects illegal control flow.
- Thread sanitizer: detects race conditions when working with multiple threads.

AFL requires example inputs to be provided in order to start it's fuzzing procedure, in order to create sample inputs, we use CSH [56]. CSH is a command line utility that can be used to interact with devices using CSP, we capture the packets sent for several commonly used commands as the initial examples.

Initial experiments using this technique turned out to be quite slow, so in the end we modified some of the code to inject packets directly into the program. This combined with switching to AFL's persistent mode, which fuzzes the target multiple times in one process, instead of only sending one packet to a freshly started program. Both of these techniques combined resulted in a 20x speed up in the fuzzing execution speed. We also slightly modified part of CSP in order to decrease the amount of packets discarded by checks such as CRC, HMAC, etc.

After each fuzzing campaign, we minimize the crash input using AFL's test case minimizer in order to find the simplest case that crashes the program. This makes it easier to identify any problems in the code.

5.2. Mitigations during the operational phase

This section describes a set of issues in the operational phase that we developed a mitigations for.

For the internal communication on board the satellite, a broadcast-based network (such as CAN) is used. The most glaring issue with broadcast-based networks is the fact that every node connected to the network can read any message sent on the network. The second issue is spoofing, which is explained in Figure 5.1 below.

In the examples below we consider a network with four nodes. There are two OBC's as well as two payload computers. However, one of the payload computers is controlled by a malicious actor. This can be through an earlier exploitation of the module or because they were one of the third-parties allowed on the satellite platform.

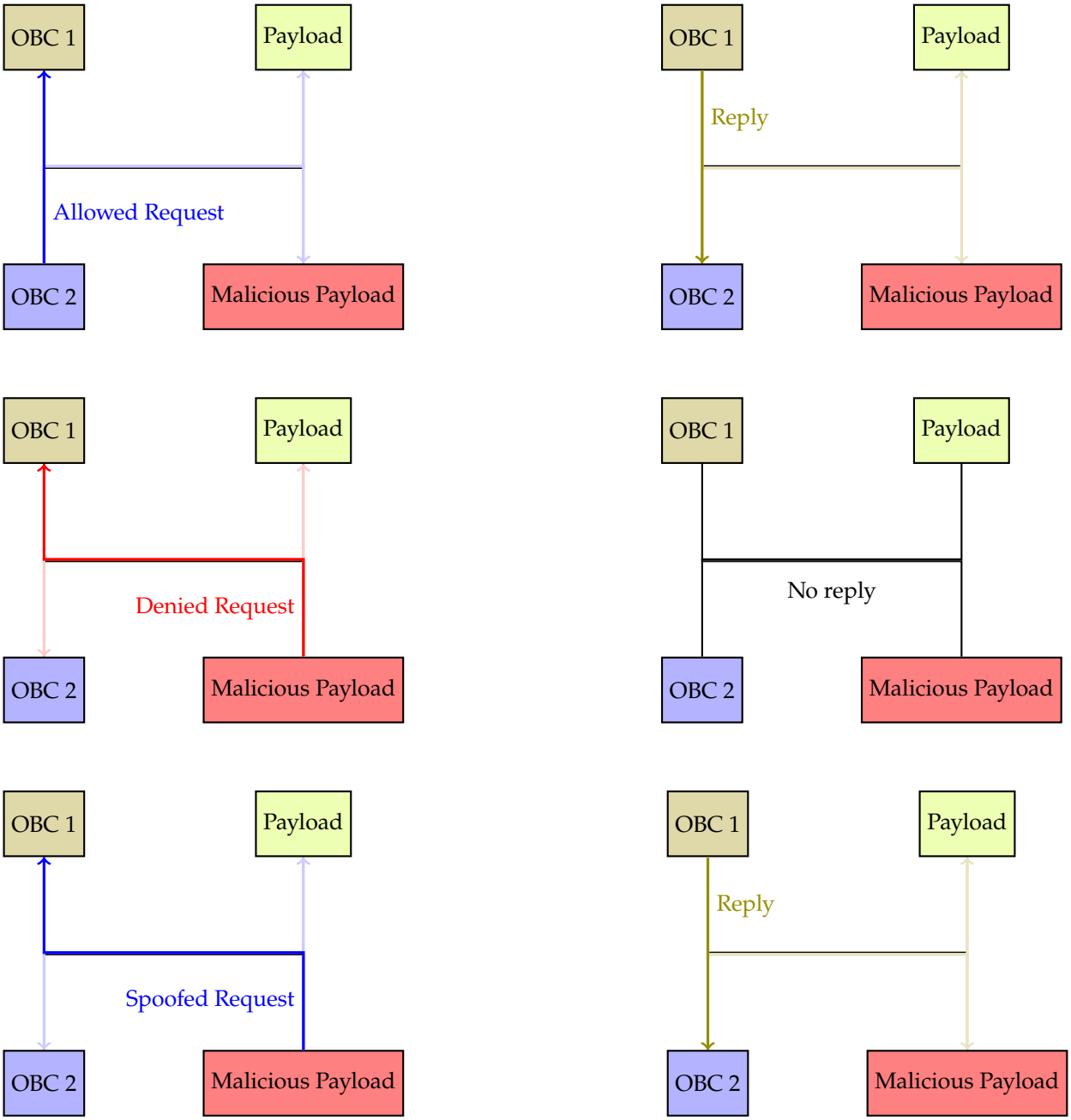


Figure 5.1: Example of spoofing on a broadcast based network

Figure 5.1 shows an example where a threat actor controlling a payload computer wishes to operate OBC 1, but OBC 1 only accepts requests from OBC 2. The malicious actor can send a CSP packet with the node ID set to OBC 2, this way OBC 1 will reply to message as normal. The malicious payload is also able to read the response that OBC 1 sends as a reply to the request.

This problem is (partially) solved by both of the mitigations listed below.

5.2.1. Security gateway

One possible solution to the above issues is a security gateway or firewall. The firewall acts as a form of access control by filtering out messages it deems not allowed. This filtering can be a simple rule based system, but also arbitrarily complex. Of course we have limited computation resources on such a device, so the filtering system should not be too computationally expensive.

Since CAN is a broadcast network, the firewall will only have effect on the boundaries between networks. These segments can be separate buses inside the system, but can also be used to filter packets coming from outside the device going in and vice-versa. We can not limit what nodes send to each other on one side of the network and we can not distinguish between devices in a single network. Thus if the gateway forwards traffic to the network with a malicious node, that node can still read the messages sent. The gateway only prevents the malicious node from seeing messages on the other side of the gateway.

In Figure 5.2 the gateway sits between two separate networks. The gateway can enforce rules on the boundaries between these two networks. Given the spoofing capability, each network should be considered it's own node when creating rules, since any node can pretend to be another node on their network. In the example, the gateway can stop a spoofed message from the malicious node, because it knows which nodes are on which side of the network, thus recognizing it as a spoofed message. However, the normal payload computer is not able to differentiate messages coming from the malicious payload and coming from the gateway (a separate network). This means that the malicious actor can pretend to send a message from the other side of the gateway (pretending to be one of the OBC's) to command the payload module.

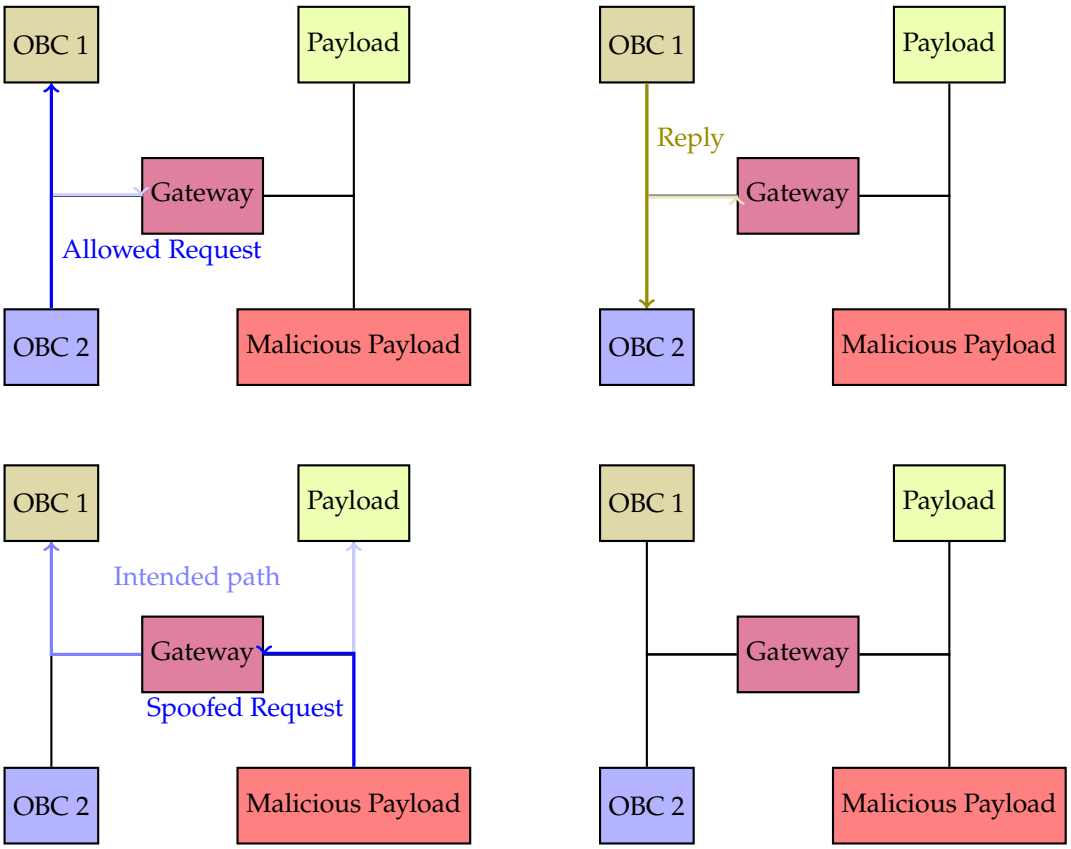


Figure 5.2: Security gateway example

The countermeasure allows the developer to e.g. only allow communication from one side of the network (OBC 1 + 2) to the other (Payload + malicious payload) or only allow certain types of messages (only pings, etc.). This way the developer/designer can isolate critical components from being affected by certain nodes, in case they are compromised. This technique can be good to limit access to a set of simple components that don't have to capability to process expensive authentication/encryption schemes.

5.2.2. Authenticated encryption

As discussed in previously, messages on a broadcast-based network such as CAN can be read by any node and nodes can easily spoof messages. An authenticated encryption is another solution to this problem.

Encryption prevents nodes from listening in on the communication of other nodes. While we use authentication to verify that a message is actually send by the node that it claims to be sent by. In order to prevent replay attacks, we use a counter in the message so that duplicate message are not processed.

With AEAD, the contents of all messages sent on the network are encrypted and thus can not be read by any node except the intended recipient. Among the associated data in the packet, is the node ID, which will be authenticated. Based on the node id, the recipient can verify the signature in the message and verify that the sender has the associated node ID, if this is not the case, the message is known to be spoofed and can be dropped.

This approach is more comprehensive in terms of the security it provides and also more flexible, since access control rules can be defined on a node level compared to the networks in the security gateway approach.

Implementation details

For this proof of concept implementation we assume pre-shared symmetric keys between each component that is communicating. The cryptographic operations were implemented using Ascon [57]. Ascon is the new NIST standard for lightweight cryptography, we use the Ascon-128 variant.

This implementation [52] proposes a header similar to IPSEC/SDLS as recommended in [23]. We use 1 byte for the SPI, which is used to differentiate the use of different encryption keys, algorithms and other rules [21], in case the underlying algorithm is ever changed for example. We use counter mode for the cypher, so we also add a three byte sequence number to the additional data. That combined with the 16 byte authentication header, we end up with a 20 byte header.

Ascon-128 uses a 64 bit (8 bytes) block size, the Initialization Vector (IV) needs to be the same size. The IV consists of a static part of five bytes and a counter (sequence number) which is three bytes. The sequence number gets incremented with every message sent. For every encryption, an internal block counter also gets increased, but this is part of the underlying implementation of Ascon.

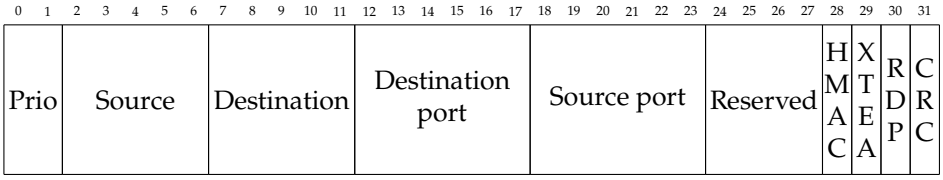


Figure 5.3: CSP version 1 header

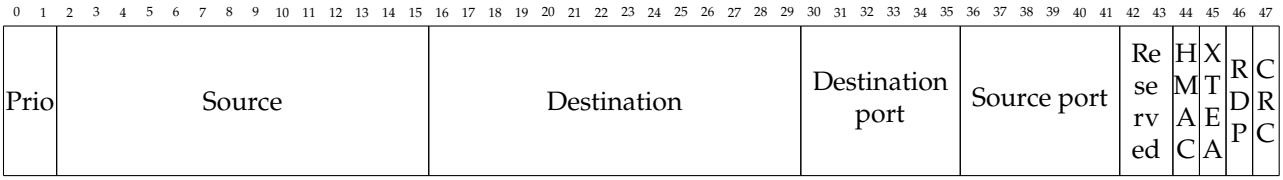


Figure 5.4: CSP version 2 header

In order to achieve the spoofing protection, we also need to authenticate the CSP header, such that the source address (node identifier) can not be changed. An overview of the AEAD scheme can be seen in Figure 5.5. Figures 5.3 and 5.4 show versions 1 and 2 of the CSP header respectively, we apply a mask to the CSP header to authenticate only the required fields. In the proposed scheme, we include all fields except for the priority field and Cyclic Redundancy Check (CRC) flag bit, in the code the mask can be configured however. The CSP header, the SPI and the sequence number are the associated data, the payload of the packet is encrypted and then both are used to produce the authentication tag. Our implementation does this step where CSP would normally to XTEA encryption and HMAC authentication, afterwards a CRC code is calculated an appended.

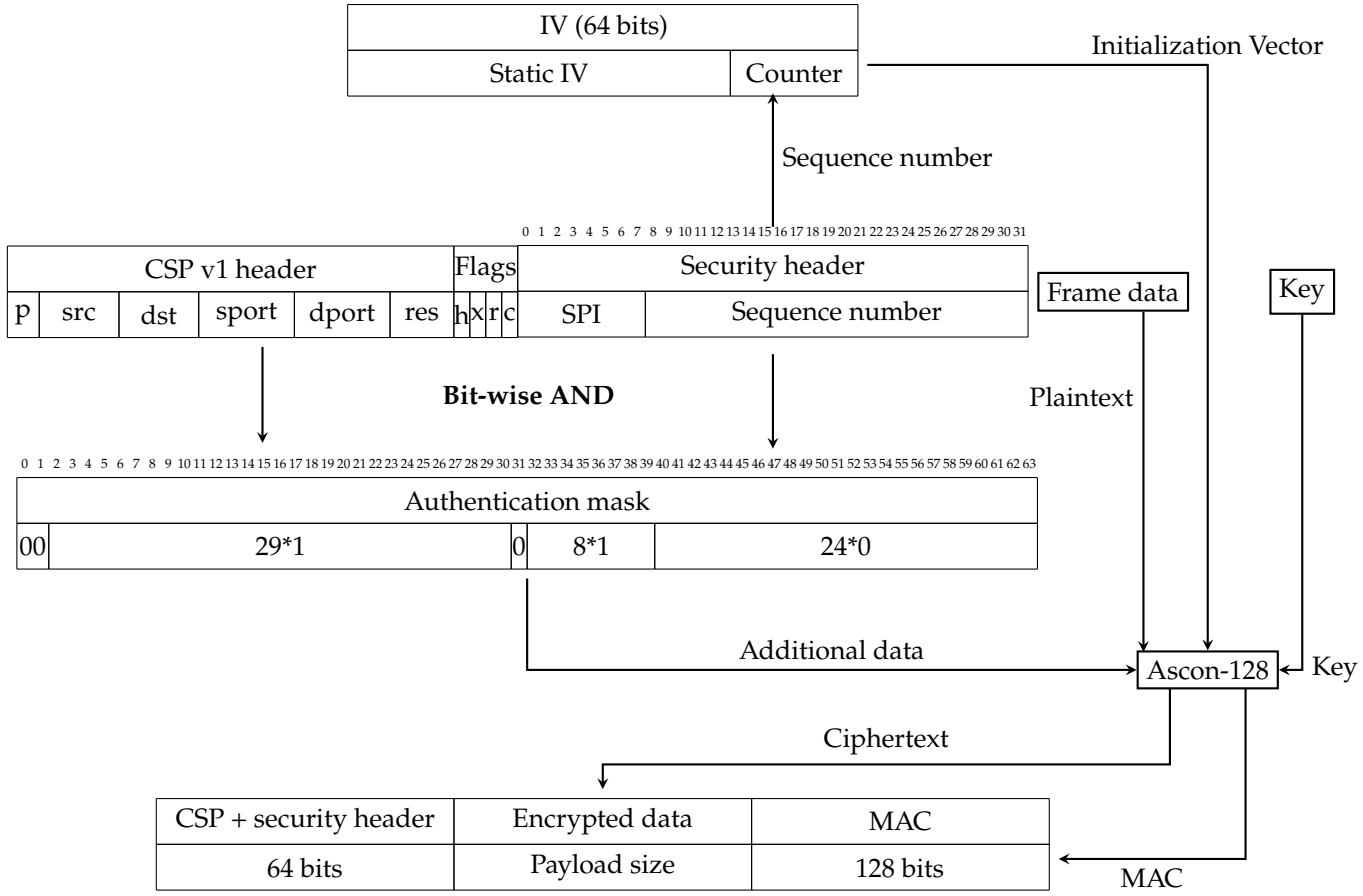


Figure 5.5: Overview of the AEAD construction

6 Results

The performance implications of these countermeasures is also measured on the real hardware setup in the lab to show how viable they are in a real scenario.

6.1. Fuzzing

In Table 6.1 and Table 6.2, one can see the results of the different fuzzing campaigns. The tables show the sanitizer/strategy used, the version of the software under evaluation and the amount of crashes found during the run. Each of the fuzzing campaigns was run for 2 hours, this was deemed sufficient as no new paths were found for a while before stopping the run. Table 6.1 shows the results of the old versions of the software and Table 6.2 uses the newer versions. Table A.1 in Appendix A shows the exact versions of the software in use.

Technique	CSP Header Version	Crashes
Address Sanitizer	v1	91
CMPlog	v1	165
Address Sanitizer	v2	110
CMPlog	v2	146

Table 6.1: Fuzzing results for old versions

Technique	CSP Header Version	Crashes
Address Sanitizer	v1	249
CMPlog	v1	1057
Address Sanitizer	v2	247
CMPlog	v2	3909

Table 6.2: Fuzzing results for newer versions

Libparam findings

The results in this section were found on both version libparam.

During fuzzing, a lot of crashes were caused by memory leaks. During the investigation we found that most of these crashes were found when libparam was interacting with MPack, one of the underlying dependencies that is used to manage serialization in the MessagePack format [58]. After further investigation, we were not able to identify any security risks with this implementation.

To null terminate string parameters in libparam, a null byte is copied at the end. Because of this, we can leak one byte of memory if the string is already at maximum size because we copy one byte more. The place in the code where this happens can be seen Figure 6.1 This is technically considered an overflow of memory, but we can not abuse this mechanism as the overflowing byte is not user controlled.

```
    """
    void param_set_string(param_t * param, const char * inbuf, int len) {
        param_set_data_nocallback(param, inbuf, len);
        /* Termination */
        if (param->vmem && param->vmem->write) {
            param->vmem->write(param->vmem, (uint32_t) (intptr_t) param->addr + len, "", 1);
        } else {
            memcpy(param->addr + len , "", 1);
        }
        /* Callback */
        if (param->callback) {
            param->callback(param, 0);
        }
    }
```

Figure 6.1: Code snippet showing where the memory leak was found

CSP findings

In the CSP library we were unable to find any direct vulnerabilities. We were able to identify a few issues that are important for developers to do to properly.

The CSP service handlers define some basic behavior can be enabled by the developer. Enabling this functionality register 2 service handlers that are relevant to consider for security.

The first being the peek and poke functionality, this allows packets to respectively read and modify memory on the device. Any packet can read confidential information on the device or read undefined memory (e.g. NULL) which crashes the OBC. Lastly, the poke functionality allows packets to directly write to memory on the device, allowing malicious actors to do practically anything.

The second issue exists only on the old version of CSP if the developer improperly configures the devices by using the incorrect API. In the CSP configuration structure, there are several fields that must be set, namely: *hostname*, *model* and *revision*. If one of these is not set, sending an *ident* (identification) command to the OBC, it will try to read one of these values and cause a segmentation fault. The newer version of CSP fixes this issue by initializing these variables with default values.

6.2. Performance evaluation of authenticated encryption

For the measurements we measure the round trip time, that means the procedure takes the following steps:

1. Encrypt packet
2. Send over CAN
3. Decrypt and verify MAC
4. Encrypt again
5. Send back over CAN
6. Decrypt and verify MAC again

We measure the time it takes to execute this procedure 200 times. A diagram explaining this procedure in more detail can be seen in Figure 6.2

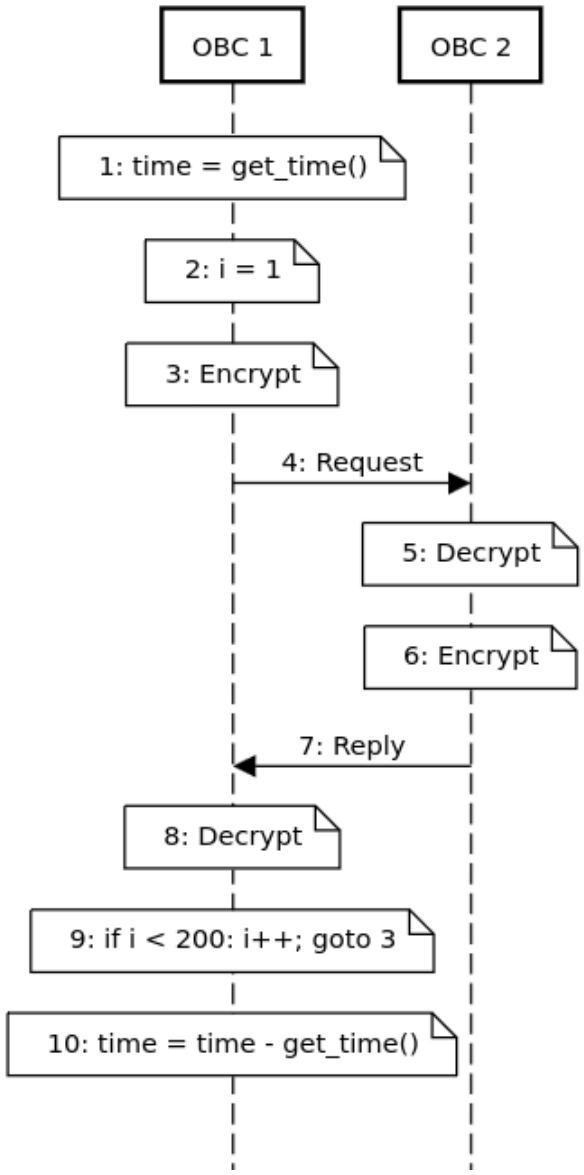


Figure 6.2: Diagram showing how the measurements were taken

We take the measurements this way because we want to enable others to think about the possibility of integrating this, which requires a relevant metric for the actual latency of receiving a reply to a message.

We measure the times of sending regular CSP packets as a baseline, we compare this to doing the full AEAD on the packet as specified earlier, as well as sending messages with additional bytes to simulate the header. We do this in order to see how much of the overhead is just for the computation of Ascon and how much is due to the increased size of the packet.

For both the full AEAD and the dummy security header, we test 3 different schemes

1. 16 bytes: just the authentication tag
2. 20 bytes: authentication tag + 4 bytes of additional data (the proposed scheme for the mitigation)
3. 32 bytes: authentication tag + 16 bytes of additional data

The results of these experiments can be found in Table 6.3, the times are in milliseconds. In Figure 6.3 one can see the overhead in % compared to not having a header.

MSG Length	No header	Dummy header 16	Dummy header 20	Dummy header 32	Ascon 16	Ascon 20	Ascon 32
50 bytes	530	650	660	760	680	690	790
100 bytes	910	1030	1080	1150	1070	1120	1190
150 bytes	1300	1420	1460	1530	1460	1510	1580
200 bytes	1710	1840	1850	1950	1890	1910	2010

Table 6.3: Overhead in ms for 200 messages

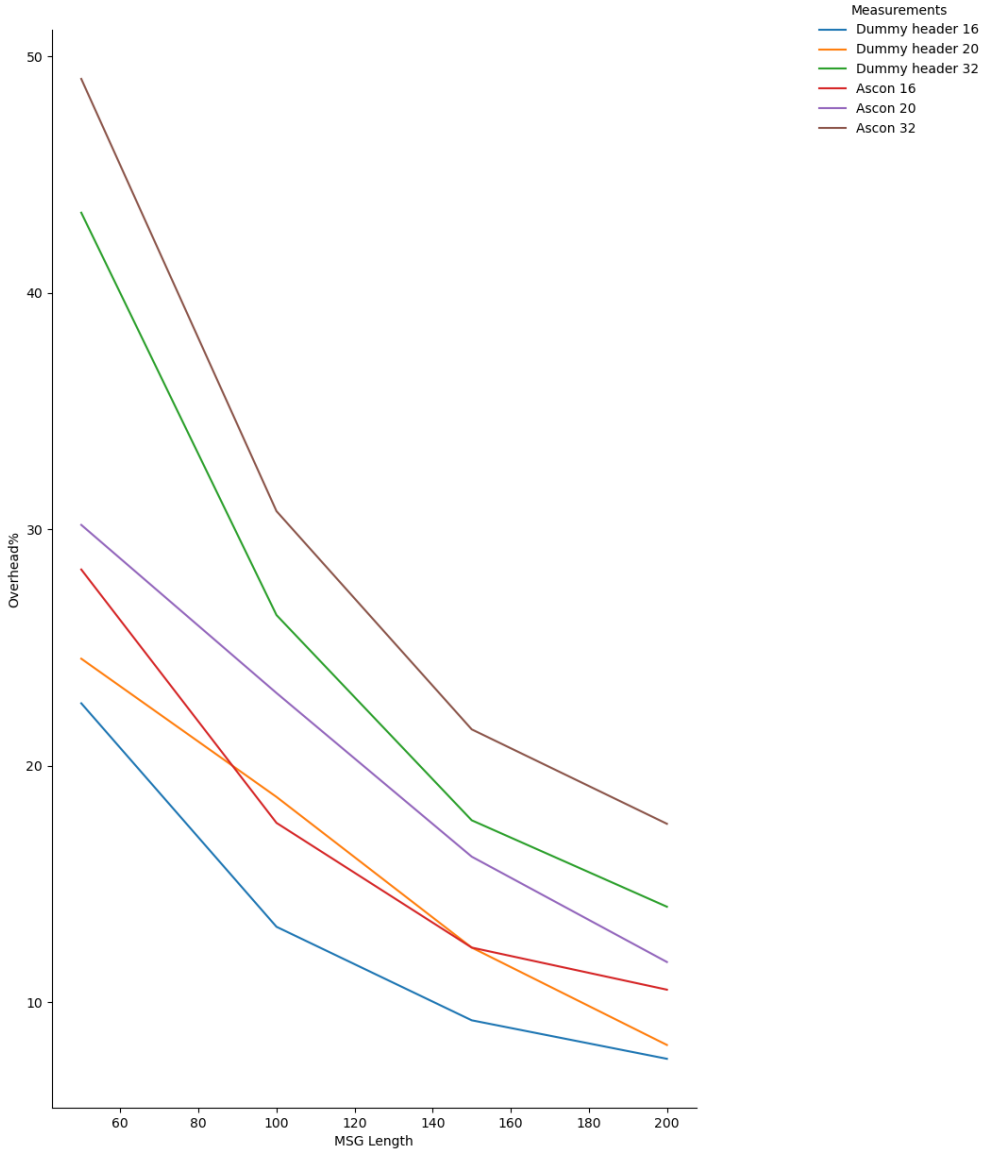


Figure 6.3: Overhead % for different header sizes

7 Discussion

This section discusses the results of the research and considerations on which mitigations of should be applied to space systems.

Fuzzing applications and especially protocols and libraries shared between multiple projects is almost always worth the time. Reducing the chance of initial access on the satellite is key in preventing security incidents. But it must not be forgotten that bypassing this outer layer is often enough to compromise the entire systems as all the components are fully trusted. Therefore defense-in-depth mitigations should be considered all systems.

Packet filtering for incoming and outgoing packets is worth doing on practically all satellites as it is very low cost to do simple rule based filtering. Including security gateways between buses can be worth doing on certain systems, but is likely to be expensive as it requires design changes. This perhaps makes it not very feasible for CubeSats, as they are meant to be affordable devices. It might be more useful to include on larger satellite systems, as there is usually a higher budget and higher security requirements. As a defense-in-depth measure, this mitigation is particularly useful for segregating components with little computational power, as the heavy lifting is done by the gateway component.

The authenticated encryption scheme prevents most of the issues with regards to internal communication and only requires changes in the software. This makes it a relatively attractive option to integrate, but the performance overhead of using it is quite significant. Most applications in space are luckily not as time sensitive as e.g. the automotive industry, so the delay is not as critically important.

Another factor to consider are the conditions in space (i.e., radiation), which can introduce faults (bit flips, damaged memory) in the on board components. These faults can often be resolved by manually manipulating memory through peek and poke (direct read and write) instructions. If such a fault would cause the authentication mechanism to fail, this would prevent authorized parties from fixing potential issues.

In the end, it really depends on the security requirements whether or not implementing AEAD is worth doing.

7.1. Implementation hurdles

The Flatsat has proven to be a difficult platform to develop for. There is a microchip on board the test bed which does not provide full access to the required source code. The developers have made workarounds through some custom linker scripts and patches. This in and of itself is not a problem, but it has caused problems for changing underlying dependencies. We failed upgrading CSP, FreeRTOS and libparam to newer versions on the device and were thus left with the old versions.

The simulator software relies on a newer version of FreeRTOS for simulation on POSIX. Therefore, we had to use two different versions of most of the libraries. This required us to port any code we wrote to the two different platforms. In order to do this, we developed a some compatibility in order to make this step easier as can be seen in the source code [52].

Fuzzing

Our initial approach to fuzzing using preeny introduced a problem with sending multiple packets in one fuzzing run. If we are not able to send multiple packets, we can not test any of connection based parts of the code, as these keep state. Normally, AFL runs the program using the given the input and then exits. In order to send multiple packets, one needs to split the input using e.g. a magic separator or the Fuzzed Data Provider (FDP) [59].

Furthermore, preeny limits the amount of different lower level network interfaces we can use. This required us to switch to the new approach after a lot of trial and error. The new approach (as described in Section 5.1.1), required us to change the underlying interface every time we wanted to test it. This cost some time, but in the end was worth it for the performance increase.

7.2. Limitations

Due to the hurdles with the implementation, the fuzzing campaigns were not as extensive as we would have liked. We would have liked to use more different sanitizers and fuzzing strategies to try and find vulnerabilities. In our research we also looked only at CSP and libparam, we would have liked to test some of other libraries and protocols as well. It would have been interesting to also consider the SPP and to verify CCSDS security standards (SDLS).

The implementation of the AEAD scheme uses the reference implementation for Ascon. This version is less performant than the optimized version for the specific hardware platform. Using the optimized version of the code should increase the performance of the implementation, resulting in less overhead. Due to time constraints we were not able to test this.

8

Conclusion

In summary, this research looked into the security of CubeSats by first analyzing potential risks for space systems and specifically CubeSats. We related all the risks to the SPACE-SHIELD framework and contributed to it wherever it was necessary. Based on this analysis, we found a set of potential vectors for compromising a CubeSat. A set of three mitigations was then implemented to improve the security by mitigating these attack vectors. These mitigations were implemented and tested on the custom simulator and later on the physical testbed.

A fuzzing campaign was applied to multiple versions of the CubeSat Space Protocol (CSP) and libparam, an associated library for managing settings on the CubeSat. This resulted in the discovery of several bugs in the tested libraries, some of which are relevant for security. These problems were also tested and verified on the physical testbed.

Two additional countermeasures were proposed for use during the operational phase to try and improve the security of CubeSats. Namely a security gateway that filters packets on the boundaries between networks and a lightweight Authenticated Encryption with Associated Data (AEAD) scheme for protecting the internal communication on the system. The source code of these mitigations is released on Github [52] for anyone to use. The performance of this AEAD scheme was evaluated in terms of latency and it showed that the overhead can range from 15-30% depending on the size of the message.

Lastly, several improvements were made to the SPACE-SHIELD framework in order to increase the knowledge base on space systems.

8.1. Future work

This research lays the ground work for further security analyses and mitigations to build upon. Some possible directions for further studies could be:

- **Point-to-point based protocols**: The FlatSat that was used during this research only had a CAN bus for internal communication. Future work could look into point-to-point based protocols such as SpaceWire/SpaceFibre and whether these technologies solve some of the problems discussed in this thesis.
- **Crytography**: This thesis only looked into one cryptographic algorithm, e.g. Ascon. We only evaluated Ascon's reference implementation and only used latency as a measure for it's performance. Future work could look into other cryptographic algorithms and metrics in order to fully evaluate which approach is the best.
- **Lower level attacks**: The implemented mitigation works on a higher layer in the Open Systems Interconnection (OSI) model, than say CAN. New Research could look into attacks on this level and whether it is able to bypass the security measures discussed in this thesis.
- **Integration with CSP**: Our implementation of the AEAD scheme is only a prototype. If one could integrate our (or a similar) implementation into protocols more tightly in order to create a reference implementation, that would allow for easier testing and research into the viability of such schemes. Such an integration should include key management and use modern cryptographic algorithms and standards.

References

[1] J. R. Kopacz, R. Herschitz, and J. Roney, "Small satellites an overview and assessment," *Acta Astronautica*, vol. 170, pp. 93–105, 2020.

[2] M. Manulis, C. P. Bridges, R. Harrison, V. Sekar, and A. Davis, "Cyber security in new space: Analysis of threats, key enabling technologies and challenges," *International Journal of Information Security*, vol. 20, pp. 287–311, 2021.

[3] J. Willbold, M. Schloegel, M. Vögele, M. Gerhardt, T. Holz, and A. Abbasi, "Space odyssey: An experimental software security analysis of satellites," in *IEEE Symposium on Security and Privacy*, 2023.

[4] J. M. Willis, R. F. Mills, L. O. Mailloux, and S. R. Graham, "Considerations for secure and resilient satellite architectures," in *2017 International Conference on Cyber Conflict (CyCon US)*, IEEE, 2017, pp. 16–22.

[5] M. Andraschko, J. Antol, S. Horan, and D. Neil, "Commercially hosted government payloads: Lessons from recent programs," in *2011 Aerospace Conference*, IEEE, 2011, pp. 1–15.

[6] N. Yadav, F. Vollmer, A.-R. Sadeghi, G. Smaragdakis, and A. Voulimeneas, "Orbital shield: Rethinking satellite security in the commercial off-the-shelf era," in *2024 Security for Space Systems (3S)*, IEEE, 2024, pp. 1–11.

[7] J. G. Oakley, *Cybersecurity for Space: Protecting the Final Frontier*. Apress, 2020.

[8] Q. Wang and S. Sawhney, "Vecure: A practical security framework to protect the can bus of vehicles," in *2014 International Conference on the Internet of Things (IOT)*, IEEE, 2014, pp. 13–18.

[9] D. Yu, R.-H. Hsu, J. Lee, and S. Lee, "Ec-svc: Secure can bus in-vehicle communications with fine-grained access control based on edge computing," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 1388–1403, 2022.

[10] F. Davoli, C. Kourogiorgas, M. Marchese, A. Panagopoulos, and F. Patrone, "Small satellites and cubesats: Survey of structures, architectures, and protocols," *International Journal of Satellite Communications and Networking*, vol. 37, no. 4, pp. 343–359, 2019.

[11] L. Mazzarella, C. Lowe, D. Lowndes, *et al.*, "Quarc: Quantum research cubesat—a constellation for quantum communication," *Cryptography*, vol. 4, no. 1, p. 7, 2020.

[12] I. F. Akyildiz and A. Kak, "The internet of space things/cubesats: A ubiquitous cyber-physical system for the connected world," *Computer Networks*, vol. 150, pp. 134–149, 2019.

[13] G. Falco, A. Viswanathan, and A. Santangelo, "Cubesat security attack tree analysis," in *2021 IEEE 8th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, IEEE, 2021, pp. 68–76.

[14] G. Falco, "Cybersecurity principles for space systems," *Journal of Aerospace Information Systems*, vol. 16, no. 2, pp. 61–70, 2019.

[15] *ESA SPACE-SHIELD*, [Online; accessed 3. May 2024], Jun. 2023. [Online]. Available: `https://spaceshield.esa.int`.

[16] Z. Sodnik, B. Furch, and H. Lutz, "Optical intersatellite communication," *IEEE journal of selected topics in quantum electronics*, vol. 16, no. 5, pp. 1051–1057, 2010.

[17] M. Bradbury, C. Maple, H. Yuan, U. I. Atmaca, and S. Cannizzaro, "Identifying attack surfaces in the evolving space industry using reference architectures," in *2020 IEEE Aerospace Conference*, IEEE, 2020, pp. 1–20.

[18] J.-L. Terraillon, *SAVOIR*, [Online; accessed 10. Jul. 2024], Nov. 2021. [Online]. Available: `https://savoir.estec.esa.int/SAVOIRMain.htm`.

[19] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices.," in *NDSS*, 2018.

[20] Space Link Protocols Working Group (SLS-SLP), *CCSDS 130.0-G-4: Overview of Space Communication Protocols*, [Online; accessed 3. Aug. 2024], Apr. 2023. [Online]. Available: `https://public.ccsds.org/Pubs/130x0g4e1.pdf`.

[21] Space Data Link Layer Security Working Group (SLS-SEA-DLS), *CCSDS 355.0-B-2: Space Data Link Security Protocol*, [Online; accessed 3. Aug. 2024], Aug. 2022. [Online]. Available: `https://public.ccsds.org/Pubs/355x0b2.pdf`.

[22] Space Data Link Layer Security Working Group (SLS-SEA-DLS), *CCSDS 355.1-B-1: Space Data Link Security Protocol - Extended Procedures*, [Online; accessed 3. Aug. 2024], Feb. 2020. [Online]. Available: `https://public.ccsds.org/Pubs/355x1b1.pdf`.

[23] , *CCSDS 355.0-G-3: The Application of Security to CCSDS Protocols*, [Online; accessed 3. Aug. 2024], Mar. 2019. [Online]. Available: `https://public.ccsds.org/Pubs/350x0g3.pdf`.

[24] M. Bozdal, M. Samie, S. Aslam, and I. Jennions, "Evaluation of can bus security challenges," *Sensors*, vol. 20, no. 8, p. 2364, 2020.

[25] C. Plummer, P. Roos, and L. Stagnaro, "Can bus as a spacecraft onboard bus," in *DASIA 2003-Data Systems In Aerospace*, vol. 532, 2003.

[26] S. Parkes and P. Armbruster, "Spacewire: A spacecraft onboard network for real-time communications," in *14th IEEE-NPSS Real Time Conference, 2005.*, IEEE, 2005, pp. 6–10.

[27] *ECSS-E-ST-50-11C – SpaceFibre – Very high-speed serial link (15 May 2019) | European Cooperation for Space Standardization*, [Online; accessed 11. Aug. 2024], May 2019. [Online]. Available: `https://ecss.nl/standard/ecss-e-st-50-11c-spacefibre-very-high-speed-serial-link`.

[28] K. Marinis and T. Szewczyk, *SAVOIR4Cubesats Workshop*, [Online; accessed 6. Aug. 2024], Oct. 2022. [Online]. Available: `https://indico.esa.int/event/429/contributions/7545`.

[29] *libcsp*, [Online; accessed 3. May 2024], May 2024. [Online]. Available: `https://github.com/libcsp/libcsp`.

[30] *FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions*, [Online; accessed 3. May 2024], Dec. 2023. [Online]. Available: `https://www.freertos.org`.

[31] *Zephyr Project – A proven RTOS ecosystem, by developers, for developers*, [Online; accessed 10. Jul. 2024], Jul. 2024. [Online]. Available: `https://www.zephyrproject.org`.

[32] H. Leppinen, "Current use of linux in spacecraft flight software," *IEEE Aerospace and Electronic Systems Magazine*, vol. 32, no. 10, pp. 4–13, 2017.

[33] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 2123–2138.

[34] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence.," in *NDSS*, vol. 19, 2019, pp. 1–15.

[35] L. Seidel, D. C. Maier, and M. Muench, "Forming faster firmware fuzzers.," in *USENIX Security Symposium*, 2023, pp. 2903–2920.

[36] F. Göhler, "Hacking the stars: A fuzzing based security assessment of cubesat firmware,"

[37] L. Seidel and J. Beier, "Bringing rust to safety-critical systems in space," in *IEEE Security for Space Systems (3S)*, May 2024.

[38] G. Marra, U. Planta, P. Wüstenberg, and A. Abbasi, "On the feasibility of cubesats application sandboxing for space missions," *arXiv preprint arXiv:2404.04127*, 2024.

[39] Y. Michalevsky and Y. Winetraub, "Spacetee: Secure and tamper-proof computing in space using cubesats," *arXiv preprint arXiv:1710.01430*, 2017.

[40] A. S. Siddiqui, Y. Gui, J. Plusquellic, and F. Saqib, "Secure communication over canbus," in *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, IEEE, 2017, pp. 1264–1267.

[41] B. Elend and T. Adamson, "Cyber security enhancing can transceivers," in *Proceedings of the 16th International CAN Conference*, 2017.

[42] A. Taylor, N. Japkowicz, and S. Leblanc, "Frequency-based anomaly detection for the automotive can bus," in *2015 World Congress on Industrial Control Systems Security (WCICSS)*, IEEE, 2015, pp. 45–49.

[43] O. Challa, G. Bhat, and J. Mcnair, "Cubesec and gndsec: A lightweight security solution for cubesat communications," 2012.

[44] S. Jackson, J. Straub, and S. Kerlin, "Exploring a novel cryptographic solution for securing small satellite communications.," *Int. J. Netw. Secur.*, vol. 20, no. 5, pp. 988–997, 2018.

[45] Systems Architecture Working Group (SEA-SA), *CCSDS 350.1-G-3: Security Threats against Space Missions*, [Online; accessed 3. Aug. 2024], Feb. 2022. [Online]. Available: `https://public.ccsds.org/Pubs/355x0b2.pdf`.

[46] D. Livingstone and P. Lewis, *Space, the Final Frontier for Cybersecurity?*. Chatham House. The Royal Institute of International Affairs, 2016.

[47] K. W. Ingols, "Design for security: Guidelines for efficient, secure small satellite computation," in *2017 IEEE MTT-S International Microwave Symposium (IMS)*, IEEE, 2017, pp. 226–228.

[48] K. Tindell, "Can bus security attacks on can bus and their mitigation," 2020.

[49] J. Pavur and I. Martinovic, "Building a launchpad for satellite cyber-security research: Lessons from 60 years of spaceflight," *Journal of Cybersecurity*, vol. 8, no. 1, tyac008, 2022.

[50] *libparam*, [Online; accessed 9. Aug. 2024], Aug. 2024. [Online]. Available: `https://github.com/spaceinventor/libparam`.

[51] K. Packard, *picolibc*, [Online; accessed 9. Aug. 2024], Jan. 2024. [Online]. Available: `https://keithp.com/picolibc`.

[52] W. Jehee, *Source code for the simulator*, [Online; accessed 12. Aug. 2024], Aug. 2024. [Online]. Available: `https://github.com/WJehee/thesis`.

[53] *slirp / libslirp · GitLab*, [Online; accessed 10. Jul. 2024], Jul. 2024. [Online]. Available: `https://gitlab.freedesktop.org/slirp/libslirp`.

[54] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, USENIX Association, Aug. 2020.

[55] *preeny*, [Online; accessed 3. May 2024], May 2024. [Online]. Available: `https://github.com/zardus/preeny`.

[56] *csh*, [Online; accessed 4. Aug. 2024], Aug. 2024. [Online]. Available: `https://github.com/spaceinventor/csh`.

[57] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon v1.2: Lightweight authenticated encryption and hashing," *J. Cryptol.*, vol. 34, no. 3, p. 33, 2021. DOI: `10.1007/S00145-021-09398-9`. [Online]. Available: `https://doi.org/10.1007/s00145-021-09398-9`.

[58] *MessagePack: It's like JSON. but fast and small.* [Online; accessed 16. Aug. 2024], Nov. 2022. [Online]. Available: `https://msgpack.org`.

[59] *The LLVM Compiler Infrastructure Project*, [Online; accessed 3. May 2024], May 2024. [Online]. Available: `https://llvm.org`.

A

Software versions

Table A.1 shows the version or truncated commit hash for each of major libraries used in this project. Some parts of the software did not use a library at all, these cases are represented by N/A (Not applicable).

Software	Physical testbed	Simulated
CSP	3ea1722f	91dc8992
FreeRTOS	v10.2.0	v11.0.1
FreeRTOS-Plus-TCP	N/A	v4.0.0
libparam	f21a54a9	ab4b5ddd
libslirp	N/A	129077f9
picolibc	5ce50345	N/A

Table A.1: Software versions for simulated and physical setup