

NATIONAL UNIVERSITY OF SCIENCE AND TECHNOLOGY POLITEHNICA BUCHAREST

FACULTY OF AUTOMATIC CONTROL AND COMPUTER SCIENCE

PARAMETER-DRIVEN GENERATION OF ATTACK-FAULT TREES FOR MULTI-TENANT ONBOARD SOFTWARE ARCHITECTURES

Coordinators,

S.I. Adina-Roxana Iordache-Munteanu
Dr.-Ing. Zain Haj Hammadeh (DLR)

Student,

Ioana-Iasmina Dănescu

Bucharest, 2026

Abstract

CubeSats, onboard edge computing, and Satellite-as-a-Service (SataaS) represent three converging trends in the space industry, spanning satellite class, technical capability, and service architecture. Together, they reflect the needs to reduce mission cost, shorten deployment time, increase onboard processing capability, and leverage reusable spacecraft infrastructure. The New Space trends suggest that spacecraft, which are inherently cyber-physical systems (CPS), are becoming more software-intensive than ever. In architectures where applications of different origin, assurance level, and criticality may share onboard computing resources, such as the one provided by DLR's (Deutsches Zentrum für Luft- und Raumfahrt) Stellar Apps project, a major safety-security problem arises: third-party or experimental software may run next to platform services that control the access to mission-critical and safety-critical spacecraft functions.

This thesis proposes a parameter-driven framework for constructing and analysing Attack-Fault Trees (AFTs) for multi-tenant onboard software (OBSW) architectures. The framework targets a class of multi-tenant architectures and is demonstrated through a SataaS use case, where the OBSW architecture is described through parameters covering trust domains, communication channels, lifecycle controls, resource isolation, and protection mechanisms. This use case is different from a terrestrial multi-tenant software architecture because of the environment in which it operates, the strong limits on CPU, storage, power, memory, lack of direct access to the system after launch, delayed or intermittent ground contact, and failure impact (mission- or safety-critical).

The metamodel of the framework defines what the generated AFTs must look like structurally to enable comparison between them. The trees are constructed from reusable fragments selected through activation rules targeting the parameters that describe the OBSW architecture of the selected use case. Architectural choices are linked to reachable attack-fault paths and, therefore, changes in these choices will produce changes in the fragments forming the final tree.

For framework demonstration, the prototype implementation compares two use case representative architecture variants for a tenant-platform privilege boundary enforcement failure top event. Structural AFT comparison is combined with semi-quantitative likelihood scoring based on attack-event feasibility and enabling fault-condition strength. The results show how design choices regarding isolation, mediation, lifecycle control and resource-control mechanisms affect the exploitable paths.

Acknowledgements

I would like to express my wholehearted gratitude to the Flight Software department within the Institute of Software Technology at the German Aerospace Center (DLR) in Braunschweig for giving me the opportunity to carry out my internship and develop my master's thesis in collaboration with DLR. I would like to thank my colleagues for making me feel welcome from the beginning. I would not have gained half of the knowledge and perspective reflected in this work without the environment, discussions, and opportunities I found here.

I am especially thankful to Dr.-Ing. Zain Haj Hammadeh for his guidance, support, valuable feedback, and the many insightful conversations in between throughout the development of this thesis. Under his supervision in the *Real-Time and Security* (RTS) group, I learned to better understand the importance of cybersecurity for space systems, a topic whose relevance I had not fully understood before. His encouragement and trust in my work helped me gain confidence in my own abilities and approach the subject with more clarity and independence.

Finally, I would like to sincerely thank my academic coordinator, S.I. Adina-Roxana Iordache-Munteanu, for her patience, availability, and willingness to answer even my smallest questions. Her dedication and genuine interest in her students made a significant difference during this process.

Contents

List of Abbreviations	v
List of Figures	vii
List of Tables	viii
List of Equations	ix
Introduction	1
1 Literature Review	5
1.1 Multi-Tenant Software Architectures in Modern Space Missions	5
1.1.1 Onboard Software Layering and Trust Boundaries	7
1.1.2 Security Implications of Shared Onboard Computing	12
1.2 Tree-Based Safety and Security Analysis	14
1.2.1 Fault Trees	14
1.2.2 Attack Trees	17
1.2.3 Attack-Fault Trees	17
1.2.4 Automated and Model-Based Tree Generation	18
1.3 Cybersecurity in Space Systems	21
1.3.1 Cybersecurity Standards and Guidance	21
1.3.2 Existing Work in Threat Modeling for Space Systems	22
1.4 Research Gap	23
2 Parameter-Driven Attack-Fault Tree Generation Framework	25
2.1 Framework Overview	25
2.2 Scope and Modelling Assumptions	26
2.3 Framework Artefacts	27
2.3.1 System Description: Parameterised Architecture Input as the AFT Driver . . .	27
2.3.2 Attack-Fault Tree Metamodel	30
2.3.3 AFT Fragment Library	38
2.3.4 Activation Rules for Fragment Instantiation	39
2.3.5 Attack-Fault Tree Generation	42
2.4 Semi-Quantitative Analysis of the Generated AFTs	42
2.4.1 AFT Comparison Across Architecture Variants	47

3 Framework Demonstration: Tenant-Platform Privilege Boundary Enforcement Failure Top Event	50
3.1 Demonstration Overview	50
3.2 Encoded Framework Artefacts	51
3.2.1 architecture_schema.yaml	51
3.2.2 architecture_variant.yaml	53
3.2.3 architecture_validator.py	54
3.2.4 aft_fragments.yaml	54
3.2.5 aft_rules.yaml	57
3.2.6 aft_generator.py	59
3.2.7 aft_visualizer.py	60
3.2.8 aft_likelihood_analyzer.py	61
3.2.9 aft_comparator.py	62
3.3 Evaluated Architecture Variants	63
3.3.1 Variant 1: Microkernel-Based VM/Container SataaS Architecture with Platform-Mediated Tenant Communication	64
3.3.2 Variant 2: Monolithic Containerized SataaS Architecture with Direct Tenant APIs and Tenant Shared Memory	65
3.4 Generated AFTs	67
3.5 Analysis and Comparison of the Generated AFTs	68
3.5.1 Structural Comparison	68
3.5.2 Likelihood Comparison	73
4 Discussion	76
4.1 Interpretation of the Demonstration Results	76
4.2 Value of Parameter-Driven AFT Generation	76
4.3 Structural Comparison versus Likelihood Comparison	77
4.4 Reusability of the Framework	77
4.5 Limitations	78
4.5.1 Framework-Level Limitations	78
4.5.2 Implementation-Level Limitations	79
Conclusion	80
References	83

List of Abbreviations

Abbreviation	Meaning
AE	Attack Event
AFT	Attack-Fault Tree
AOCS	Attitude and Orbit Control System
API	Application Programming Interface
BFM	Boundary Failure Mode
CAN	Controller Area Network
CAPEC	Common Attack Pattern Enumeration and Classification
cFS	Core Flight System
COTS	Commercial-Off-The-Shelf
CPS	Cyber-Physical System
CSP	Cubesat Space Protocol
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DAG	Directed Acyclic Graph
DoS	Denial-of-Service
DSL	Domain-Specific Language
ECSS	European Cooperation for Space Standardization
EPS	Electrical Power System
FC	Fault Condition
FDIR	Fault Detection, Isolation, and Recovery
FE	Failure Event
FMEA	Failure Mode and Effects Analysis
FMECA	Failure Mode, Effects and Criticality Analysis

Continued on next page

Abbreviation	Meaning
FTA	Fault-Tree Analysis
MTG	Meteosat Third Generation
NMF	NanoSat MO Framework
OBC	On-board Computer
OBSW	Onboard Software
OS	Operating System
PUS	Packet Utilization Standard
RTOS	Real-Time Operating System
SAND	Sequential AND
SataaS	Satellite-as-a-Service
SFTA	Software Fault Tree Analysis
SPARTA	Space Attack Research & Tactic Analysis
TC	Telecommand
TCB	Trusted Computing Base
TE	Top Event
TM	Telemetry
VM	Virtual Machine

List of Figures

1.1	Multi-tenancy across cloud service layers	6
1.2	Conceptual representation of multi-tenancy in SataaS	6
1.3	Layered OBSW architecture of the MTG satellite platform	8
1.4	Layered Stellar Apps OBSW architecture	10
1.5	Conceptual positioning of OBSW architecture references with respect to flexibility and explicit treatment of multi-tenancy	11
1.6	Simple fault tree structure	15
1.7	Example of attack-fault tree	18
2.1	Framework process diagram	26
2.2	Isolation design space	29
2.3	Proposed attack-fault tree metamodel	32
3.1	Comparator-supported architecture refinement process	63
3.2	Variant 1: Microkernel-Based VM/Container SataaS Architecture with Platform-Mediated Tenant Communication	65
3.3	Variant 2: Monolithic Containerized SataaS Architecture with Direct Tenant APIs and Tenant Shared Memory	67
3.4	Generated trust-material exposure BFM for Variant 1	71
3.5	Generated trust-material exposure BFM for Variant 2	72

List of Tables

2.1	Example of parameterised architecture variants	28
2.2	Activation Logic of Event Types	41
2.3	Attack Event Likelihood Scoring	44
2.4	Fault Condition Likelihood Scoring	45
2.5	Failure Event Likelihood Score Derivation	46
3.1	Architecture parameters encoded in the framework demonstration	52
3.2	Representative allowed values from <code>architecture_schema.yaml</code>	52
3.3	Number of FEs and CWEs defined per BFM in the prototype implementation	55
3.4	Generated AFT size for the evaluated architecture variants	68
3.5	Structural comparison of generated AFTs by BFM family	69
3.6	Top-event likelihood comparison	73
3.7	BFM likelihood comparison	74

List of Equations

2.1	Joint probability of an FE	43
2.2	Conditional probability expression for FE likelihood	43
2.3	Raw FE likelihood score	45
2.4	OR-gate probability for two input events	47
2.5	Ordinal OR-gate likelihood aggregation	47
2.6	BFM likelihood aggregation	47
2.7	Top event likelihood aggregation	47

Introduction

Background and Context

For most of the space age, cybersecurity, compared to other mission assurance concerns such as reliability, availability, and protection against environmental hazards, received less explicit engineering attention throughout the mission lifecycle [1]. Satellites were often treated as closed and highly specialised systems, operated through proprietary infrastructures and partly shielded by the physical remoteness of space. This resulted in dominant risks being associated with component malfunction, design faults, radiation effects, or other environmental disturbances, rather than deliberate malicious actions. Software was primarily viewed as a supporting part of the spacecraft system, rather than as an attack surface in its own right. These assumptions, however, no longer reflect the reality of contemporary space missions.

Modern societies rely heavily on space infrastructure for navigation, timing, communication, Earth observation, environmental monitoring, and defence [1]. Critical services, ranging from financial transactions to aviation, logistics, emergency response, and military operations depend on satellite-enabled capabilities, often in ways that remain invisible to end users.

The societal dependence on satellites has reshaped the threat landscape, making them more attractive targets because their compromise can provide strategic leverage and large-scale disruptive effects. Unlike kinetic attacks, which may generate debris and are more easily attributable, cyberattacks can be conducted remotely, at comparatively lower cost, and with a higher degree of deniability [1]. Recent incidents demonstrate that these threats are no longer only theoretical: the cyberattack on the Viasat KA-SAT network in February 2022 disrupted satellite internet services across Europe, with a major impact on Ukraine and on both military and civilian infrastructure. Satellite navigation systems have also been affected by jamming and spoofing during geopolitical tensions, with consequences for aviation and maritime operations [2].

The rise of commercial constellations, CubeSats, onboard edge computing, and Satellite-as-a-Service (SataaS) concepts is transforming satellites into software-intensive and interconnected cyber-physical systems (CPS). These missions rely on complex onboard software (OBSW), commercial off-the-shelf (COTS) components, cloud-connected ground infrastructures, reusable platform services, and, in some cases, third-party applications running on shared on-board computers (OBCs) [3], [4]. While these developments lower entry barriers and accelerate innovation, they also expand the attack surface and introduce new classes of vulnerabilities [5].

For example, Stellar Apps, a project currently being developed by the German Aerospace Center–Deutsches Zentrum für Luft- und Raumfahrt (DLR)—aims to provide an onboard execution environment for future space missions, allowing third-party applications and system services to be deployed as onboard apps [6]. These applications are supposed to execute in isolation and communicate through platform-provided interfaces, within pre-determined boundaries and resource quotas guarded by control and monitoring mechanisms. Further access of these apps to spacecraft subsystems, payload sensors or ground segment are mediated through the platform services to diminish as much as possible potential mission damage. While this approach can improve flexibility and reusability, it also creates new trust boundaries between external applications, platform software, and spacecraft-facing functions.

This architectural evolution creates a combined safety-security challenge. Traditional safety analysis methods, such as Fault Tree Analysis (FTA), are designed to help identify the ways in which an undesired system state may occur, from accidental faults and component degradation to environmental disturbances. Security methods, such as attack trees, model attacker goals and exploitation paths, but they do not connect them to system-level failures caused by faults, unsafe interactions, or architectural weaknesses. Therefore, each method captures only part of the problem.

Modern multi-tenant OBSW architectures require a perspective that considers both accidental failures and deliberate attacks, as well as their possible interactions. In some cases, the same weakness may contribute to a system fault under nominal conditions or be deliberately exploited by an adversary. For example, allocating resources without limits or throttling, identified as a common weakness during system design [7], may cause service unavailability under excessive load or unintended usage patterns. The same condition may also be exploited intentionally to trigger a denial-of-service (DoS) scenario.

Attack-Fault Trees (AFTs) provide one suitable modelling basis for this need by combining elements of fault trees and attack trees within a single model. They make it possible to represent how accidental fault conditions and malicious attack events may jointly contribute to system-level failure. This makes them useful for software-intensive space systems in which reliability and security cannot be treated as fully independent properties.

Unlike a multi-tenant platform, which may host applications from multiple customers alongside trusted platform services on the same OBC, a traditional single-operator satellite has fewer organisational trust boundaries and a more stable set of operational assumptions. This is because all the software on the single-operator satellite comes from one organisation, goes through one development and assurance process, and is verified as a complete integrated system before launch.

As discussed earlier, tenant applications are not intended to hold direct authority over safety-critical or mission-critical spacecraft functions, meaning that their requests should be mediated by platform-provided mechanisms. The effectiveness of this mediation depends on the selected OBSW architecture, including its isolation mechanisms, communication policies, shared-resource controls, lifecycle management, and access-control enforcement.

Problem Statement

The challenge is to understand how each differently configured architecture changes the ways in which a system-level failure may occur.

Selecting AFTs as the modelling formalism is only part of a potential solution. They provide a structure that combines both accidental faults and deliberate attacks, but the construction of an AFT for a concrete architecture remains dependent on manual modelling. The resulting tree is influenced by the analyst's experience and assumptions, limiting reproducibility and reducing confidence that the constructed tree is consistently aligned with the system boundaries and system components under analysis.

In the SataaS context considered in this thesis, operators may need to evaluate several architectural configurations during early design, differing in choices such as tenant isolation, tenant-to-tenant communication, platform mediation etc. If AFTs are constructed manually for each variant, the comparison may become inconsistent, as differences between trees may reflect modelling decisions rather than actual architectural differences. Besides the problem of them being potentially inconsistent across different system configurations, there is also the issue of the AFT construction being difficult to repeat with each change in the architecture.

The idea of linking architectural choices to safety and security risks appears in space software engineering guidance. The European Cooperation for Space Standardization (ECSS) Space Engineering standard ECSS-E-ST-40C Rev.1 requires software security analysis to be extended and revised iteratively during architectural definition, and requires the impact of the selected architecture on identified security risks to be documented at Preliminary Design Review (PDR) stage [8]. It also requires the selected architecture to be reviewed to ensure that it addresses identified security risks and that design components do not introduce additional unacceptable risks.

Generating security analysis outputs from system descriptions has been demonstrated in terrestrial software engineering through work like model-based threat analysis, which generates possible threats from architecture descriptions [9]. Moreover, ATSyRA supports attack tree synthesis from system descriptions expressed through a domain-specific language (DSL) [10]. Both show that structuring the system inputs can reduce manual effort and improve consistency, but neither targets space systems and neither works with AFTs.

Research Question and Objectives

The thesis is guided by the following research question:

How can architecture variants for multi-tenant OBSW be compared before implementation to support informed protection decisions for higher-criticality software components, while accounting for the fact that protection mechanisms may close some attack-fault paths and introduce or reshape others?

To address this question, the thesis pursues five objectives by moving from the representation of multi-tenant OBSW architectures to the construction of architecture-specific AFTs, and then to the comparison of the generated tree fragments.

Objective 1: Define a parameterised representation of multi-tenant OBSW architectures.

The first objective is to describe architecture variants through a set of parameters. Since the analysis is intended for early design, the input must include architectural properties before detailed implementation information is available. This is the first step to providing a common basis for analysing and comparing different architecture variants.

Objective 2: Define an AFT metamodel for architecture-driven generation.

The second objective is to establish the event types, hierarchy, and decomposition logic used by generated AFTs. A fixed metamodel is needed so that generated trees follow the same modelling conventions instead of reflecting arbitrary analyst choices, making the resulting AFTs easier to interpret and compare across architecture configurations and selected top events.

Objective 3: Develop reusable AFT fragments and activation rules.

The third objective is to define reusable AFT fragments, which represent sub-trees and are derived with respect to one top event. Instead of constructing each tree from scratch, the method uses fragments that can be selected and combined according to the architecture variant that is being analysed. The activation rules are the ones to trace these fragments to the architecture parameters and determine which are structurally applicable to a given variant. This objective supports repeatable construction and reduces dependence on subjective decisions.

Objective 4: Define a comparison and semi-quantitative analysis method.

The fourth objective is to compare generated AFTs across architecture variants not only structurally, but also quantitatively. Since detailed probability data are usually unavailable during preliminary design, the analysis uses ordinal likelihood scores based on (i) attack-event feasibility and enabling (ii) fault-condition strength.

Objective 5: Demonstrate feasibility through a prototype and case study.

The final objective is to show that the defined elements can be combined in a coherent workflow. The prototype encodes the framework artefacts and applies them to representative SataaS OBSW architecture configurations. The purpose of the demonstration is not to prove completeness for all possible multi-tenant OBSW architectures, but to assess the feasibility and internal consistency of the approach.

Each of these objectives leads to the main outputs of the thesis: a parameterised architecture schema, an AFT metamodel, a reusable fragment library with activation rules, a comparison and scoring method, and a prototype-based SataaS case study.

Chapter 1

Literature Review

1.1 Multi-Tenant Software Architectures in Modern Space Missions

Missions and platforms such as OPS-SAT-1 and DPhi Space show how spacecraft can now support software experiments or hosted payloads by allowing external access to onboard capabilities [11], [12].

COTS components and small-satellite buses, together with the provision of reusable, modular platform onboard software stack and the possibility of post-launch, on-orbit software updates help change spacecraft from a fixed-function asset into a CPS whose capabilities may evolve throughout its operational life [13].

Satellite-as-a-Service, or SataaS, refers to a business model in which satellite owners or operators provide selected spacecraft capabilities to external users through controlled access mechanisms. This model is popular in commercial providers because reusing the same spacecraft infrastructure across users or missions reduces the cost and development effort associated with designing, testing, launching, and operating dedicated satellites [14]. The term is broader than one specific implementation, meaning that these capabilities may vary, depending on the mission concept, from allowing access to data products, payload services, onboard processing, software experimentation, communication services, or controlled use of onboard resources.

The analogy with cloud service models is helpful because both cases involve reusable infrastructure and shared resources, but only if the layer at which sharing occurs is explicitly mentioned. In cloud computing, the meaning of multi-tenancy changes depending on whether tenants share infrastructure, a runtime platform, an application, or a database service.

In Infrastructure-as-a-Service (IaaS), tenants share physical hardware while typically running separate operating system (OS) instances. In Platform-as-a-Service (PaaS), tenants share an OS, runtime platform, or application environment while deploying their own applications. In Software-as-a-Service (SaaS), tenants share an application instance and are primarily separated at the data or configuration layer. In Database-as-a-Service (DBaaS), tenants share a database service while maintaining logically distinct configurations, schemas, or datasets [15].

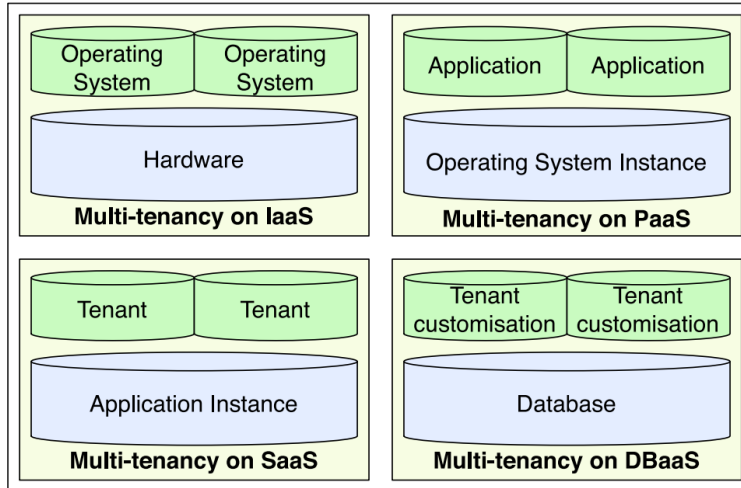


Figure 1.1: *Multi-tenancy across cloud service layers*

Source: [15].

SataaS does not fit cleanly into any of these categories. Its closest analogue is PaaS, since multiple tenant applications may execute on a shared OBC. However, the analogy remains incomplete because the shared platform is not only a runtime environment. In SataaS, platform services may also mediate access to spacecraft-facing functions, which means that the tenant separation boundary is also a safety-security boundary.

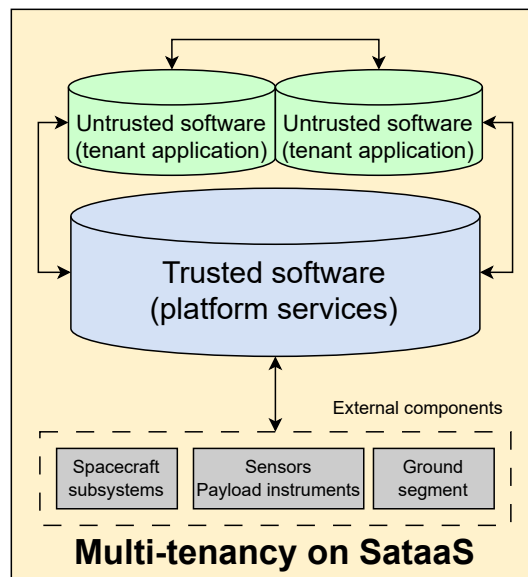


Figure 1.2: *Conceptual representation of multi-tenancy in SataaS, where tenant applications access spacecraft-facing functions through trusted platform services*

The conceptual structure shown in Figure 1.2 highlights this difference. Tenant applications

may share onboard computing resources, but they should not interact directly with mission-critical or safety-critical spacecraft functions. Access to payload operation, attitude-related services, power-relevant resources, data handling, or TM/TC-related functions must instead pass through trusted platform services. These services are responsible for authorising requests, enforcing limits, controlling communication, and preventing tenant behaviour from propagating beyond its permitted scope.

OPS-SAT [11] and ROSPIN-SAT-1 [16], for example, cannot be inherently considered SataaS missions because they are framed as experimental laboratories or open-access testbeds, rather than providers of services to certain customers. However, they are worth mentioning here because they demonstrate the technical principle of allowing third-party entities to access satellite functions within a controlled environment.

Industry examples such as Exodus Orbitals and DPhi Space are closer to SataaS in the stricter service sense. Exodus Orbitals is explicitly described as a SataaS platform that allows businesses and developers to host and run satellite applications in space without requiring costly dedicated hardware [14]. Similarly, DPhi Space advertises hosting payloads on a Linux-based shared platform, deploying them as Docker containers in orbit. [12].

The Stellar Apps project is the closest reference, from an OBSW architecture perspective, for the specific mission context addressed in this thesis. While Exodus Orbitals and DPhi Space help define the service model, Stellar Apps helps define the exact architectural problem: externally supplied applications executing on a shared OBC, managed through lifecycle controls, and granted configurable access to spacecraft resources through trusted platform services [6].

1.1.1 Onboard Software Layering and Trust Boundaries

The software architecture onboard a spacecraft dictates how mission and platform functions are allocated to software components, how those components interface with onboard hardware and communication buses, and how the system supports data exchange, command handling, and runtime coordination. It has a system-level role in spacecraft operation because it implements complex requirements and provides the main interface through which ground operators control and supervise the spacecraft [17].

The Meteosat Third Generation (MTG) satellite platform provides a useful starting point because it shows a conventional operational OBSW architecture with explicit layers and component roles. Its software is organised into vertical layers and transversal layers: the vertical layers include the Application Layer, the Services Layer, and the Low Level SW Layer, while the transversal layers include the OBSW Framework and the Basic Services transversal layer [17]. The Application Layer contains spacecraft and mission management functions, including mode management, subsystem management, health-resource control, FDIR (Fault Detection, Isolation, and Recovery), and on-board control procedures. The Services Layer provides common data-handling and PUS (Packet Utilization Standard) services. Within this layer, the Hardware Abstraction sub-layer groups the controllers used to access OBC input/output boards and communication buses. The Low Level SW

Layer contains the RTOS (real-time operating system) and hardware driver software. The OBSW Framework provides shared mechanisms such as messaging, access to a central data repository, and operating system (OS) abstraction through the OSAL. The Basic Services transversal layer contains generic reusable objects such as queues, lists, data pools, and mathematical libraries [17].

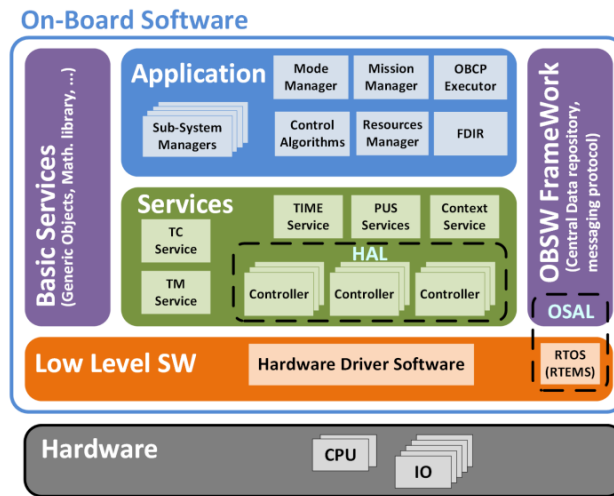


Figure 1.3: Layered OBSW architecture of the MTG satellite platform

Source: [17].

MTG has components separated according to architectural responsibility: mission-level functions are placed in the Application Layer, common services and hardware-facing controllers are placed in the Services Layer, low-level software handles RTOS and driver interaction, and transversal framework mechanisms support communication and data access across components. For a parameter-driven method, this type of structure shows where architectural parameters can be attached: to application components, service layers, hardware-facing controllers, shared repositories, message paths, and OS interfaces.

The core Flight System (cFS) adds a reusable flight-software perspective to the same discussion, by defining API-based layers that separate the OS and board support package from the core Flight Executive and the application layer [18]. The Operating System Abstraction Layer and Platform Support Package provide platform-independent access to OS and hardware services, while the core Flight Executive provides common runtime services for flight applications. These runtime services include the Software Bus for inter-application messaging, time management, event messages, table management, and executive services for startup and runtime control [18].

cFS treats applications as managed software components, as opposed to indistinct parts of one flight software image. Applications execute within a strict runtime environment and interact through core Flight Executive services. Since resources are managed per application, cFS can support operations such as starting, stopping, and loading individual applications during runtime, providing an architectural precedent for later multi-tenant reasoning: once applications are managed separately, the analysis can ask which services they use, which messages they exchange,

and which runtime mechanisms control their interaction with the rest of the spacecraft software.

MTG and cFS establish the baseline for the architectures discussed next: MTG shows how an operational spacecraft OBSW stack can separate mission functions, common services, hardware abstraction, while cFS shows how reusable flight software can organise applications around shared runtime services and a software bus.

More recent OBSW approaches extend layered and component-based design toward application-oriented software platforms. In these architectures, onboard applications are developed and managed as separate software entities, while platform services provide the interfaces needed for monitoring, control, communication, and access to spacecraft resources. SataaS and multi-tenant OBSW can only be managed safely if the platform defines how they are deployed, updated, started, stopped, monitored, and allowed to interact with other onboard functions.

The NanoSat MO Framework (NMF) is an early example of this app-oriented view of OBSW. It was developed around the idea of turning onboard software into flexible applications that can be developed, debugged, tested, deployed, and updated through defined interfaces [19]. In the NMF, an app can access platform peripherals through platform services and can also be started, monitored, stopped, killed, installed, uninstalled, and updated from ground [19]. Apps are allowed to expose monitoring and control services to ground systems or to other onboard apps, supporting interaction between software elements without requiring each app to handle low-level spacecraft details directly. The NMF Supervisor allows multiple apps to share the same platform services for interaction with onboard peripherals. In this architecture, platform services form the interface between applications and spacecraft resources. Applications are separated from low-level implementation details and the platform retains control over how onboard resources are accessed, creating a clearer boundary between application logic and platform authority, which later becomes important for trust and security reasoning.

Application-oriented OBSW management systems follow a similar direction from the perspective of micro- and nanosatellite software management. Recent work describes traditional flight software as highly customised, tightly coupled to hardware, and limited in adaptability, then proposes an open software management system based on Linux, standardised interfaces, middleware components, communication services, and application modules [20]. In this type of architecture, application modules are treated as managed software objects, while the management system handles scheduling, communication, and hardware-interface access. The support for on-orbit updates and dynamic reconfiguration is needed for missions that need to add, replace, or update onboard processing functions during operation [20].

Cloud-native satellite computing extends the same architectural direction by applying concepts such as containers, microservices, resource virtualisation, and service orchestration to onboard computing. In the BUPT-1 cloud-native satellite case study, applications are packaged as containers or services and uploaded to onboard computing components [21]. The paper distinguishes between a cloud-native load, where payload functions are packaged as services using technologies such as Docker, and a cloud-native platform, where platform equipment and interfaces are exposed through software-defined mechanisms [21]. This approach supports flexible application deployment, easier migration across hardware, and more adaptable onboard comput-

ing services.

DLR's Stellar Apps is the closest architectural reference for the type of OBSW considered in this thesis. The previous examples introduce several relevant ideas separately: NMF treats OBSW as apps with lifecycle operations, application-oriented management systems describe managed software modules and dynamic refactoring, and cloud-native satellite computing introduces containers, microservices, and software-defined platform interfaces. Stellar Apps brings these ideas into a setting where third-party applications and system services are deployed as apps, executed in isolated environments, and granted configurable access to spacecraft and computing resources [6]. The platform also includes communication via API with hardware, application updates, multi-tenant operation, resource limits, lifecycle cybersecurity measures, and explicit permissions for access to spacecraft resources [6].

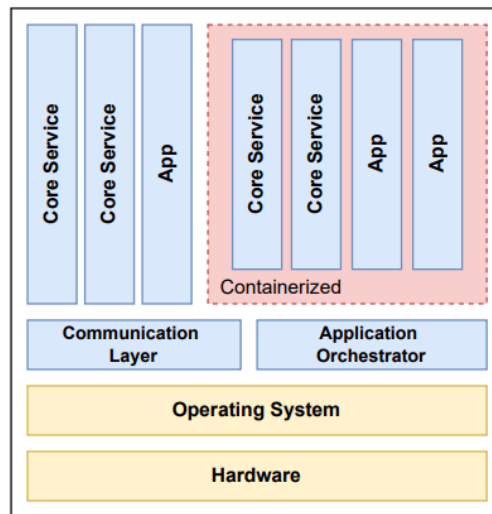


Figure 1.4: *Layered Stellar Apps OBSW architecture, illustrating platform-managed application execution, communication mediation, and containerized isolation for external applications*

Source: [6].

The reviewed references can be positioned along two dimensions that matter for this thesis: how far the architecture supports flexible onboard application management, and how explicitly it addresses trust boundaries created by third-party or tenant software. This positioning clarifies why Stellar Apps is the closest reference for the proposed framework, while the other examples support specific parts of the architectural background.

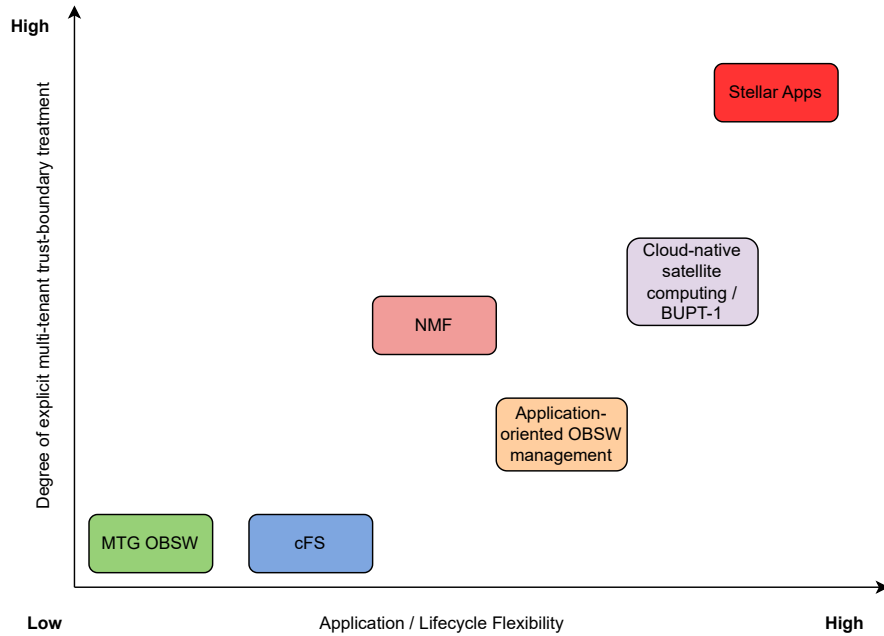


Figure 1.5: *Conceptual positioning of OBSW architecture references with respect to flexibility and explicit treatment of multi-tenancy*

Although these examples differ in implementation choices, they share recurring architectural elements. This observation is consistent with architecture modelling approaches such as AADL, which describe embedded systems through component abstractions, interfaces, interactions, flows, and properties [22]. AADL gives engineers a common vocabulary for describing a system architecture when implementation details are only partially specified. Architecture-based design method supports the same idea, but applies it to satellite OBSW using CubETH nanosatellite CDMS software case study [23]. Its purpose is design assurance, with the authors being concerned with the fact that satellite OBSW is concurrent: several software components run conceptually in parallel, share buses, memory, buffers, and services, and can create resource contention or deadlocks. They argue that verifying all possible behaviours after the design is difficult, so they use architecture patterns that enforce required properties during design. They first look at satellite OBSW and identify recurring coordination patterns, then formalise these as architecture styles, including mutual exclusion, client-server, action flow, failure monitoring, mode management, buffer management, event monitoring, and priority management [23]. Each style describes a reusable way of coordinating components and the property it enforces.

The architectures discussed above also introduce the concepts of software trust, software criticality, and trust boundaries. Software trust refers to the degree of confidence that a software component will behave acceptably for its intended needs. This confidence is not based only on the function performed by the software, but also on the information available about its development, verification, configuration control, update process, and protection against both accidental flaws and malicious insertions [24]. In multi-tenant OBSW, this distinction matters because tenant applications

originate outside the spacecraft operator’s controlled development and assurance process, while platform services remain responsible for mediating access to spacecraft resources.

According to the ECSS, software criticality refers to the consequence of a software function failing, behaving incorrectly, or being misused. In the OBSW context, an experimental image-processing application or tenant data-processing task does not have the same mission or safety relevance as software involved in attitude control, power management, TM/TC handling, FDIR, payload operation, or access to spacecraft-facing platform services. ECSS treats software criticality as a driver for engineering and assurance activities, since the required level of verification, validation, and product assurance depends on the consequences of software failure or incorrect behaviour [8], [25].

Trust and criticality describe complementary qualities of software. A trusted software component can be non-critical, while an untrusted component can still be critical if it can influence mission-critical or safety-critical functions. In multi-tenant OBSW, this distinction matters because tenant applications may have lower trust than platform services due to their external origin or limited assurance evidence, while their criticality depends on what they can reach through the architecture.

1.1.2 Security Implications of Shared Onboard Computing

Shared onboard computing changes the security assumptions of OBSW because several software elements may depend on the same processor, OS, middleware, communication services, update mechanisms, and platform APIs. In a multi-tenant OBSW architecture, some of these elements may come from external users or organisations, while trusted platform services still control access to spacecraft resources. A weakness in one application can therefore matter beyond its own function if requests, data, or faults can reach services implementing higher-criticality functions.

Modular OBSW design still needs runtime isolation. Reusable flight software frameworks such as cFS were introduced because many missions need common onboard functions, and the cFE provides a portable runtime environment with shared services for flight applications. Modularity from a development perspective does not, however, guarantee separation during execution. Jero et al. note that existing modular flight software was not originally designed around isolation between modules or basic security principles, and that cFS modules may run as threads in a shared address space [26].

Common implementation practices in satellite software reinforce this problem: widespread use of languages such as C and C++ with limited use of memory-safety mitigations, software running with excessive privileges, and OS configurations where separation between user space and kernel space may be optional or depend on configuration [26]. These practices affect the strength of the boundary between tenant applications and platform services. If applications share address space, privileges, or execution context too broadly, a local fault or compromise can become a platform-level problem.

Platform mediation is one of the main security concerns in shared onboard computing. Tenant applications should not directly command or influence spacecraft-facing functions such as AOCS

(attitude and orbit control system) services, EPS (electrical power system) resources, payload operation, or TM/TC (telemetry/telecommand) handling. Requests from tenant applications have to pass through platform services that check identity, permissions, inputs, timing, and resource use. Incomplete, overly permissive, or bypassable checks mean that tenant-originated behaviour can move beyond the scope the architecture intends. Empirical satellite firmware studies have found weaknesses along these lines, including poor access protection, unsafe command paths, arbitrary TC execution, arbitrary code execution, and DoS conditions [13].

COTS hardware and software add to this exposure. They reduce cost and development time, which explains their use in many small satellite projects, but they can also bring known terrestrial vulnerabilities into the spacecraft platform [27]. Service-oriented and reprogrammable spacecraft then add the possibility of changing onboard software after launch. That flexibility is what makes SataaS and app-oriented missions viable, but it also makes software origin, update validation, execution permissions, and access to platform resources part of the security argument.

On-orbit updates are needed to fix defects, patch vulnerabilities, add functions, and adapt the mission after launch. In multi-tenant or hosted-payload settings, the update path may cross organisational trust boundaries. Cubedate addresses this for multi-tenant CubeSats, where the OBC and the hosted payload may be operated by different stakeholders that do not necessarily trust each other [28]. In that context, the update mechanism is both a maintenance capability and a possible attack path, so update authenticity, integrity, freshness, rollback behaviour, and compatibility with the target environment all become part of the security boundary.

Hosted payloads and small satellites may have limited communication windows, limited power, and update paths that depend on the operator's ground infrastructure, making the design of on-orbit updates and recovery more difficult [28]. Patching can also be slow because updated images have to be uploaded over constrained links, with upload time depending on ground-station access and link quality [13]. A failed or malicious update might be difficult to correct quickly, and in shared on-board computing, lifecycle control directly affects whether trust in OBSW can be maintained after launch.

When several applications share a processor, OS, middleware, communication layer etc., a compromise in one should remain contained. Sandboxing confines software activity so that a vulnerability in one application does not automatically expose the rest of the system [29]. The CubeSat sandboxing study treats isolation as a practical security mechanism for small satellite software and shows that it can reduce the risk of a successful exploit leading to full platform control. The same study notes that isolation mechanisms have to be selected carefully, since containers, virtual machines (VMs), and application sandboxes differ in overhead, hardware support, integration effort, and isolation strength.

Shared resources create an availability problem. Tenant applications may compete for CPU time, memory, storage, bus bandwidth, communication capacity, power, or access to common platform services. Without quotas, scheduling controls, watchdogs, one application can degrade the availability of others or of platform functions. On a spacecraft, DoS can affect command handling, payload operation, data availability, communication paths, or services used by higher-criticality functions. Orbital Shield discusses these risks on satellite platform communication buses, including

overload and buffer-related scenarios in communication based on CAN (Controller Area Network) [27].

Communication mechanisms can carry failures or attacks across the software stack in the same way. Software buses, middleware, shared repositories, CAN links, communication over CSP (Cubesat Space Protocol), API calls, and app-to-platform interfaces make OBSW modular, but they can also carry malformed messages, excessive requests, corrupted data, or unauthorised commands if validation and access control are weak. In a multi-tenant architecture, the communication design influences how far the effect of a compromised or faulty application can travel.

As seen here, risk paths often combine deliberate actions with enabling weaknesses in the architecture. A malicious request, unsafe update, or resource-abuse action becomes dangerous when permissions are too broad, isolation is weak, update validation is insufficient, rate limits are missing, resource control is poor, or platform mediation can be bypassed. The same weaknesses can matter without a malicious actor, through misconfiguration, overload, or ordinary software faults.

1.2 Tree-Based Safety and Security Analysis

Tree-based analysis methods represent undesired outcomes by decomposing them into lower-level events, conditions, or actions. A system-level failure, compromise, or hazard is placed at the root of the tree, while the lower levels show how it can occur through alternative or combined paths. This structure makes the progression from local events to system-level consequences explicit.

This section reviews three related formalisms. Fault trees model accidental failures and their contribution to an undesired system state. Attack trees model attacker goals and the steps through which those goals can be achieved. Attack-fault trees combine both views in a single structure, allowing accidental faults and malicious actions to be represented together. This progression matches the type of reasoning needed for shared onboard computing, where tenant-originated behaviour, software faults, weak isolation, and platform-level controls may interact within the same architecture.

Although these methods are commonly presented as tree-based models, their underlying logical structure can also be represented as a directed acyclic graph (DAG) when events or sub-structures are shared by several parent nodes [30]. In a tree-oriented representation, repeated concepts are usually duplicated under each relevant branch. In a DAG representation, the same event or condition can be represented once and connected to multiple higher-level events. This can reduce duplication, while requiring clearer semantics for shared-node interpretation, visualisation, and quantitative or qualitative propagation.

1.2.1 Fault Trees

FTA is a top-down, deductive safety analysis method. It starts from a predefined undesired system state, called the top event, and decomposes it into the lower-level faults that can lead to it

[31]. If the top event is too broad, the tree can become difficult to manage; if it is too narrow, the analysis may lose the system-level view needed for safety reasoning.

A fault tree is built around one selected top event. It does not catalogue every possible failure of the system, but rather represents the credible combinations of events that can contribute to the selected undesired state, as judged within the system boundary, operating context, and level of detail chosen for the analysis [31]. The system boundary, the limit of resolution, the operating assumptions, and the available knowledge all shape what events appear in the tree.

Intermediate events (represented in rectangles) describe higher-level fault states that require further decomposition, while basic (represented in circles) or undeveloped events (represented in a diamond shape) appear at the lower levels when the analysis has reached its selected level of detail or when further information is unavailable, respectively. Gates describe the logical relationship between child events and their parent. OR gates represent alternative paths to a parent event, while AND gates represent combinations of events that must occur together [31]. Other gate types, such as inhibit or priority gates, can be used when additional conditions or ordering constraints need to be captured, but the tree remains a logical decomposition of the top event.

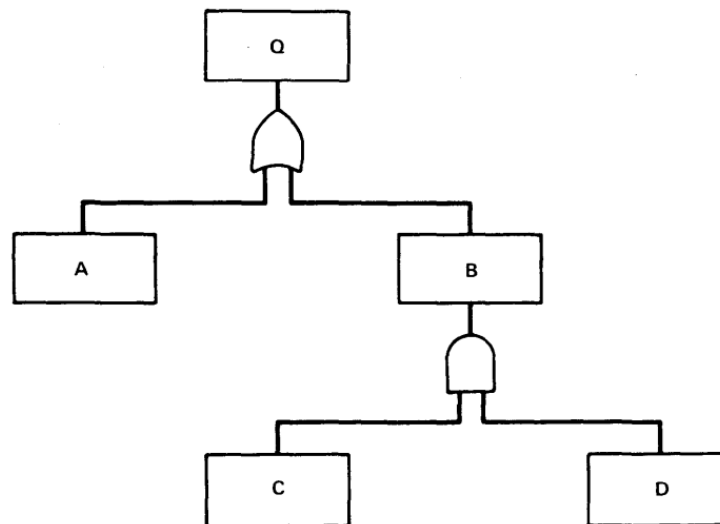


Figure 1.6: Simple fault tree structure with “Q” as top event, intermediate events “A” and “B” decomposed via an OR logic gate, and intermediate events “C” and “D” decomposed via an AND logic gate

Source: [31].

The term fault should be understood more broadly than hardware failure. The Fault Tree Handbook distinguishes failures from more general fault events: a component may fail, but a system may also enter an undesired state because a component operates correctly at the wrong time, receives an incorrect command, or is affected by an external condition [31]. Fault classification work makes the same point from a different angle. Faults may be physical, environmental, operational, hardware-related, software-related, internal, external, permanent, transient, accidental, or intentional, and their classification depends on the system, technology, environment, and applica-

tion in which they occur [32]. A fault tree can therefore include software faults, operational faults, environmental disturbances, configuration errors, command faults, or other conditions alongside failed hardware.

Fault trees can be evaluated qualitatively or quantitatively. The tree is first a qualitative logical model, even when later quantified using event probabilities [31]. Qualitative evaluation commonly focuses on minimal cut sets, the smallest combinations of basic events sufficient to produce the top event. A single-event cut set identifies a single point of failure, while higher-order cut sets show combinations of events that defeat the system together. This type of analysis can support design reasoning without probability data, because it shows which parts of the system structure make the top event reachable.

Quantitative FTA adds probability values to events and propagates them through the tree, but this depends on the availability and quality of data. For rare events, novel systems, or early design stages, many probabilities may be uncertain or unavailable. Recent work on fault trees with limited data addresses this by combining uncertainty quantification with importance metrics, allowing analysts to identify critical nodes while expressing confidence in those rankings [33]. This is relevant for early-stage safety analysis because exact probabilities are often unavailable when architecture-level decisions still need to be made.

FTA has also been applied in space-system design at low cost. In the Delfi-C3 nanosatellite project, FTA was introduced after the PDR as a qualitative method for improving reliability discussions, evaluating design changes, and identifying design flaws while modifications were still possible [34]. The analysis focused on catastrophic and critical failures rather than all possible mission degradations, because those were the cases where costly design measures such as redundancy could be justified [34]. Single points of failure and minimum cut sets of order two were used to assess the design, identify possible common-cause issues, and prioritise further design and verification work.

FTA can be applied directly to software behaviour, an approach known as Software Fault Tree Analysis (SFTA). SFTA has been used to detect software logic errors, identify conditions under which fail-safe or fault-tolerance mechanisms should be initiated, and support the generation of test cases [35]. It is typically performed qualitatively because software-related failure probabilities are difficult to estimate without suitable data for the software elements [35]. Model-based and formal-specification-based approaches show that fault trees can be constructed from software requirements or design models, which is relevant for systems where software behaviour and timing constraints contribute to safety-relevant outcomes [35]. ECSS-Q-HB-80-03A clause 6.3 recognises SFTA as a standards-level technique for analysing how software-caused failures contribute to system-level top events, applicable at the architectural design level to support software criticality analysis and component isolation decisions [25].

Fault trees provide a structured way to describe how accidental faults, operational conditions, software errors, and component failures can combine toward an undesired system state. Their limitation in the present context is that they do not model an adversary's goal or the deliberate steps taken to reach it. Shared onboard computing involves both safety-relevant failure propagation and security-relevant tenant behaviour, so fault trees provide only part of the modelling background.

1.2.2 Attack Trees

Attack trees are tree-based security models used to represent how an attacker can achieve their goal. The top event represents the main attacker objective, intermediate nodes represent sub-goals, and leaf nodes represent basic attack actions that are not further decomposed. The model keeps the hierarchical structure of fault trees, but changes the modelling perspective from accidental failure causation to adversarial goal decomposition [36], [37]. Compared to fault trees, attack-trees may use a refined logic AND gate called sequential AND, usually denoted as SAND, to express cases where all child actions are required and must be performed in a specific order [37].

Attack trees can represent technical, physical, and human-related attack steps within the same model [37]. This is one reason they have been adopted for security risk assessment across domains such as CPS, automotive systems, industrial systems, and critical infrastructure. Their structure also supports qualitative and quantitative analysis. Qualitatively, the tree can be inspected to identify alternative attack paths, repeated patterns, or critical combinations of actions. Quantitatively, attributes such as cost, time, difficulty, likelihood, required skill, or attacker effort can be assigned to basic actions and propagated toward the root through the tree structure [37].

Attack trees have also been applied in the space domain. Falco et al. use attack tree analysis to assess security of an abstracted architecture in an operational CubeSat. Their analysis considers attacker goals such as DoS, data tampering, and disabling the CubeSat, and uses attack trees to show possible routes toward those goals [38]. This provides a spacecraft-specific example of attack-tree reasoning, although the focus remains on attacker pathways rather than combined safety-security propagation.

Attack trees therefore provide the security-analysis counterpart to fault trees. They model how an adversary may reach a goal through alternative or combined attack steps, and they support both manual reasoning and automated generation from system models, vulnerability knowledge, or reusable templates. Their limitation for the present thesis is that they do not naturally represent accidental faults, enabling failure conditions, or safety-oriented propagation unless they are extended or combined with another model.

1.2.3 Attack-Fault Trees

Attack-fault trees (AFTs) are used when a system-level disruption may result from accidental failures, malicious actions, or a combination of both. This makes them suitable for CPS, where software, communication links, physical processes, and control functions interact within the same operational environment [30].

One of the early motivations for combining attack trees and fault trees was the introduction of information and communication technologies into industrial control and critical infrastructure systems. Fovino et al. argue that modern infrastructures cannot be analysed only through random component failures, since cyber vulnerabilities may allow an attacker to trigger events that also appear in a fault tree [39]. Their integration principle is that the goal of an attack tree can be treated as an event in a fault tree. If a successful attack produces a condition already represented in the

fault tree, the corresponding attack tree can be connected to that event.

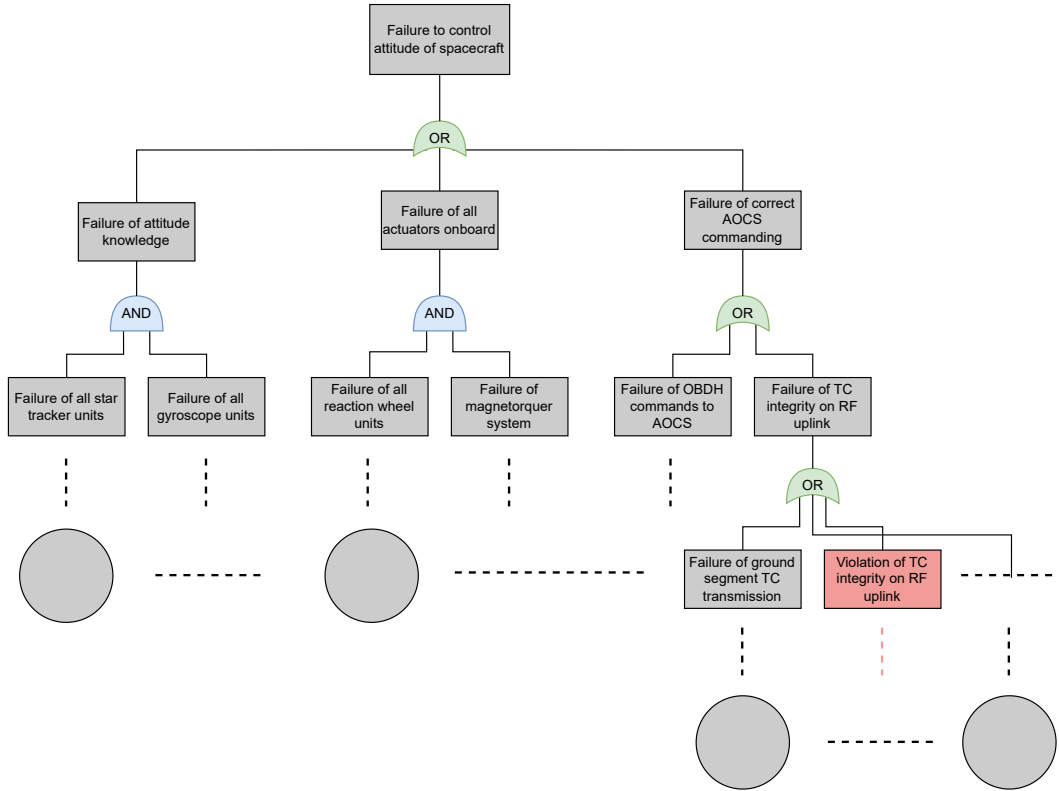


Figure 1.7: *Example of attack-fault tree integration for a spacecraft attitude-control failure, showing a cyber-related telecommand integrity violation as one contributor to a fault-tree branch*

This integration changes the interpretation of some lower-level events. A fault-tree event may be caused by an accidental failure, such as a component fault, or by a malicious action, such as disabling a protection mechanism, compromising a service, or manipulating a communication path. The same top event can therefore be reached through accidental paths, malicious paths, or mixed paths in which a failure condition enables an attack or an attacker exploits an existing weakness.

Quantitative work focuses on the relation between safety and security metrics. AFTs were analysed through Markov decision processes and studies the Pareto relation between reliability, as a safety metric, and attack cost, as a security metric [40]. This work shows that AFT analysis depends on the chosen semantics and on assumptions about what the attacker can observe before acting. AFT analysis may need to compare several metrics at once, because the most likely attack-fault path is not necessarily the cheapest or easiest one for the attacker.

1.2.4 Automated and Model-Based Tree Generation

A concern in itself is how these tree models are constructed. For small examples, manual construction may be sufficient, but in many other cases manual modelling can become slow, in-

consistent, and difficult to keep aligned with the actual system design. Automated tree generation addresses this by deriving attack or attack-fault structures from explicit inputs, instead of relying only on expert judgment during the whole tree construction process.

One survey adds value to the existing literature on actual automatic generation of tree-based methods by organising them into three main categories: model-driven, analysis-driven, vulnerability driven generation [41]. These categories are not mutually exclusive, so one paper may belong to more than just one category. Model-driven generation transforms a system description directly into an attack tree or graph using rules, such as *if an asset is in a location, and an attacker can reach that location, then a path to that asset exists*. Analysis-driven generation first analyses a system model, for example through model checking, planning, reachability analysis, or logical inference, and then use the analysis results to build the attack model. Vulnerability-driven generation starts from known vulnerabilities, reusable templates, or attack patterns then matches them to the analysed system. The system still matters, because the vulnerability pattern must be instantiated for the system under study, but the reusable knowledge comes from outside the system model.

This classification shows that automatic generation is rarely based on a single source of information. Most approaches require some combination of system structure, attacker objectives, security properties, vulnerability knowledge, reusable templates, and human assumptions. The same survey also reports practical limitations in the literature, including limited reproducibility, incomplete tool availability, and limited evidence about scalability for real systems [41]. Automation improves coverage and consistency, but it still depends on the quality of the input model and on the rules used to decide what should be generated.

One line of work generates attack trees from formal or semi-formal system descriptions [42]. Vigo et al. propose a static-analysis approach in which attack trees are inferred from a process algebraic specification [42]. The authors start from the idea that manual attack trees depend heavily on expert judgement and can become incomplete for large systems. Their solution is to generate the tree from a formal process description of the system, rather than asking the analyst to manually list attack steps. They use a process algebraic specification, specifically the Value-Passing Quality Calculus, to describe how system processes communicate, what checks they perform, and which labelled program points can be reached. This approach needs a formal process algebraic model of the system. That gives a strong semantic basis, but it may be hard to apply in early engineering contexts where the system is described through architecture diagrams, requirements, interface tables, or design parameters rather than a full formal process specification.

ATSyRa follows an analysis-driven generation with explicit user guidance: the tool starts from a high-level system description, generates attack scenarios through model checking, and then synthesizes an attack tree using high-level actions supplied by the analyst [10]. In the ATSyRA tool, the analysed system is a military building described through zones, accesses, items, alarms, attacker strength, and attacker objective. This description is compiled into an attack graph, and model checking is then used to identify sequences of atomic actions that allow the attacker to reach the objective. The authors note that naive full automation can lead to trees that are too flat and difficult to use, so the analyst helps structure the result by defining meaningful action abstractions. This work is relevant because it shows that automated attack tree generation still needs modelling

choices about abstraction and readability to match the level at which the system is analysed.

Bryans et al. are doing template instantiation over a network architecture model [43]. The analyst provides two things: a graph-like architecture description and a library of attack templates. The generator then fills template variables with real network elements and recursively expands matching leaves until no more templates apply. The network model contains subnetworks, components, attacker-accessible nodes, and a mapping showing which component belongs to which subnetwork. The idea is that a template is a parameterised attack-tree pattern: for example, a generic template may say *Eavesdrop X:NET*, where X can be replaced by any real subnetwork in the model, or *Spoof X:ECU*, where X can be replaced by a real electronic control unit. In summary, the method is about matching reusable attack patterns to architecture elements. Bryans et al. show that reusable attack knowledge can be written once as parameterised templates and then instantiated automatically according to the architecture being analysed.

Semi-automatic augmentation also combines reusable knowledge with expert control. Jhawar et al. propose starting from an initial attack tree and augmenting it with subtrees from an annotated library [44]. Their workflow starts with first manually building an initial attack tree from expert knowledge and CAPEC (Common Attack Pattern Enumeration and Classification) knowledge base. Then each node is formally annotated with assumptions and guarantees, describing when that attack step is applicable, for example which platform, product, or environment must be present. A guarantee describes what becomes true if that attack step succeeds, for example that a remote attacker can execute arbitrary code or overwrite a file. These annotations make it possible for the tool to decide whether a library subtree fits a node in the existing tree. The next step in the process described in this paper is to build a library of small attack trees from NVD/CVE data. NVD, the U.S National Vulnerability Database, is maintained by NIST (National Institute of Standards and Technology) and is built upon and fully synchronized with the CVE (Common Vulnerabilities and Exposures) list so that any updates to CVE appear here [45]. Jhawar et al. describes each library entry as a small tree representing the action (exploit CVE-YYYY-XXXX) [44]. Their tool checks whether a library tree can be attached by verifying against a rule. For example, if a node in the original tree requires “remote attacker executes arbitrary command”, then a CVE-based library tree that guarantees that action may be attached there. The main takeaway from this paper is that attack-tree generation can benefit from a hybrid approach: experts define the high-level structure, while reusable vulnerability knowledge is attached automatically only when explicit applicability conditions are satisfied.

One paper explores fragment-based generation from public vulnerability and weaknesses knowledge, where fragments are defined as small partial attack trees [46]. Instead of generating one large attack tree at once, the fragments serve as building blocks that can later be connected to larger attack chains using the relationships between CWEs (Common Weakness Enumeration), such as one weakness requiring another or one weakness preceding another. First, the generator has to be given a list of CPEs (Common Platform Enumeration) representing the software products and libraries used in the system. It then uses this list to search public vulnerability data and find the CVEs associated with that product. CWEs are used to group the CVEs by weakness type, because, as described by The MITRE Corporation [47], the relationship between CVEs and CWEs is that the latter, if present when designing and developing a system and not mitigated before deployment,

can lead to known vulnerabilities of the released product. CVSS (Common Vulnerability Scoring System) is attached to each CVE to provide severity-related metrics, while EPSS (Exploit Prediction Scoring System) estimated the probability that a CVE will be exploited over a time window [46]. This work suggests that vulnerability and weakness databases can be used to generate attack-tree fragments automatically, but this works best when the analysed system already has known software products, libraries, and versions.

MBTA: A Model-Based Threat Analysis Approach proposes the idea that an analyst can model the software architecture first, then predefined threat rules are checked automatically against the model [9]. The architecture is described using a component-port-connector style modelling language, and threats are formalised as OCL (Object Constraint Language) constraints. For example, the paper formalises threats such as Man-in-the-Middle and Injection. A Man-in-the-Middle threat is associated with missing or weak encryption, authenticity, and integrity mechanisms, while an Injection threat is associated with missing or weak input validation. These informal threat signatures are translated into OCL rules that can be checked on the architecture model. The interesting part is that the authors refine the architecture metamodel to reduce false positives. At first, their rules detect too many threats because the model does not contain enough architectural context. Then they add concepts such as trust level and port kind. Trust level helps distinguish trusted from untrusted zones, while port kind helps distinguish external/public ports from internal/private ports, improving the precision in their example.

Taken together, these works show that automated tree generation is most useful when reusable modelling knowledge is connected to an explicit description of the system under analysis. Formal approaches provide strong semantic grounding, template and library-based approaches improve reuse, vulnerability-driven approaches link generated structures to public security knowledge, and model-based threat analysis shows that architecture properties can be checked automatically against formalised threat conditions. However, the reviewed approaches also show a remaining gap. Some depend on detailed formal specifications, others require known products and CVEs, and others detect threats without constructing a safety-security causal model.

1.3 Cybersecurity in Space Systems

1.3.1 Cybersecurity Standards and Guidance

Cybersecurity guidance for space systems should follow the mission lifecycle phases, as described in ECSS-E-ST-80C. According to this, it should be considered starting from mission definition and up until operation and disposal, because threats, assets, and acceptable risk levels all change as the project matures [48]. This standard defines security principles, such as attack surface reduction, defence in depth, domain separation, fail secure, least privilege, need-to-know (NTK), secure evolvability, separation of duties, and redundancy. For multi-tenant OBSW, these principles map directly onto architectural decisions: domain separation and least privilege determine whether tenant applications are isolated from each other and from platform services, while defence in depth and fail secure behaviour concern whether the architecture limits how far a fault

or attack can travel once it occurs. The same logic applies when zooming in on the software. ECSS-E-ST-40C Rev.1 mentions a security process covering security analysis, risk treatment, and security activities across the software lifecycle specifically[8].

BSI TR-03184 space-specific security guidance focuses on information security for the space segment and links threats to protection goals and security measures, with mission-specific tailoring expected from the user [49]. Space Attack Research & Tactic Analysis (SPARTA) taxonomy adds a threat knowledge perspective for space systems, with entries covering flight software exploitation, malicious software, third-party software, lack of bus segregation, OS attacks, and critical subsystem attacks [50].

The Consultative Committee for Space Data Systems (CCSDS) also provides an informational Green Book, CCSDS 350.1-G-3, to support mission planners in identifying and understanding security threats against space missions. It provides a high-level overview of threat sources, common threats, vulnerabilities, mission impacts, and possible security mechanisms across space and ground mission infrastructure [51].

MITRE maintains widely recognized cybersecurity knowledge bases. CVE provides identifiers for publicly disclosed vulnerability instances (a specific product, version, or configuration), allowing tools, databases, and analysts to refer to the same known issue consistently [52]. CWE sits one level higher, describing classes of software and hardware weaknesses that may lead to vulnerabilities once the product containing the weakness is released; examples include missing authentication, improper access control, weak input validation, or unsafe resource management [47]. A CVE record can often be mapped to one or more CWE categories to describe the underlying weakness that caused the vulnerability. CAPEC complements both CVEs and CWEs by describing common attack patterns, meaning recurring ways in which adversaries exploit weaknesses maliciously [53].

1.3.2 Existing Work in Threat Modeling for Space Systems

Space systems have specific security conditions that affect how threats should be identified and modelled: long mission lifetimes, limited physical access after launch, constrained onboard resources, dependence on ground segment, and the safety consequences of compromised spacecraft functions. Existing work in space-system threat modeling therefore helps define the domain-specific context in which the framework proposed in this thesis is implemented. Moreover, it shows which aspects are already addressed by current methods, such as asset identification, scenario generation, and lifecycle security guidance, providing the basis for identifying the more specific gap addressed in this thesis: the lack of a parameter-driven method for generating architecture-specific AFTs for multi-tenant OBSW.

The tendency in existing work is to first decompose the space system into more specific segments, such as ground, launch, link, space, user segments, to conduct threat modeling based on the characteristics of each.

Sheik et al. examine threat modelling for New Space systems through a teleoperation use

case [54]. Their study uses a reference architecture and data flow diagrams to decompose a tele-operated robot system into hardware and software components, then applies STRIDE/DREAD to identify and rank threats across trust domains. The work is relevant because it explicitly addresses cyber-physical and hardware-software interactions, which are difficult to capture with threat-modelling approaches originally developed for conventional information systems. The authors identify telecommunications as a highly exposed asset in the use case and highlight jamming, spoofing, and Man-in-the-Middle attacks as severe threats.

Other work expands threat analysis by focusing on scenario generation. Lin et al. propose the ICARUS (Imagining Cyberattacks to Anticipate Risks Unique to Space) matrix as a scenario generator for space cyberattacks [1]. Rather than limiting the discussion to familiar examples, the report encourages analysts to generate richer scenarios by considering the attacker, target, method, segment, motivation, and timing of an attack. This report suggests the idea that many plausible scenarios may be missed if analysts only reuse familiar attack examples.

Secure design work places threat modeling within the broader lifecycle of space systems. Hammadeh et al. argue that cybersecurity should be integrated throughout the space-system lifecycle, together with secure engineering, offensive testing, cyber resiliency, and standardization [3]. Their V-model mapping shows that threat landscape analysis, security engineering, testing, operation, and resiliency need to be considered as connected activities rather than isolated checks. This means that threat modeling can support early architectural decisions, guide testing, and inform mitigation planning.

1.4 Research Gap

Space cybersecurity literature divides the space system into more specific assets or segments: ground segment, communication/link segment, space segment, user segment, sometimes launch/ cloud/supply chain too. A lot of existing work looks at the outer attack surface of those segments: jamming, spoofing, compromised ground station, malicious commands, user-terminal compromise, cloud services, supply chain, and so on. All of this is important, but then the question becomes:

Once a threat reaches the spacecraft or originates from software already onboard, what determines whether it remains contained or propagates toward spacecraft-facing functions?

The answer is the OBSW architecture, which is the internal structure of the space segment that decides how software behaviour can affect spacecraft functions. The space segment alone is not a sufficient abstraction level if the concern is the safety-security propagation inside the spacecraft.

Existing work recognises the space segment as a cyber target, and existing generation work shows that attacks or threats can be generated from models, templates, databases, or architecture descriptions. However, these ideas have not yet been applied to the internal architecture of

multi-tenant OBSW, where design parameters such as isolation, mediation, lifecycle control, authority binding, communication paths, and resource sharing determine whether tenant-originated behaviour can affect high criticality functions or not. In other words, the literature gives all the pieces, but does not combine them for this specific problem.

Chapter 2

Parameter-Driven Attack-Fault Tree Generation Framework

2.1 Framework Overview

The proposed framework supports the early comparison of shared OBSW architecture variants through parameter-driven AFT generation. It is meant to be used during the preliminary design stage of the mission, when exact implementation details such as libraries, exact OS version, and test results are not yet known. Consequently, CVEs or implementation bugs cannot be linked to the system; only structural choices such as isolation mechanisms, communication between software components and security policies like resource quotas, and control of on-orbit updates can describe the system at this design stage. What the framework does is take these choices (architectural parameters) and ask *Given these architecture conditions, which attack-fault paths are possible for this selected top event?*

The top event represents an undesired system-level failure state identified before applying the framework, for example through a system-level FMEA (Failure Mode and Effects Analysis), FTA, or mission consequence assessment.

The process has two main phases: a qualitative analysis phase and a semi-quantitative analysis phase. The qualitative analysis is the generation of AFTs itself using activated fragments based on rules derived from different combinations of the parameters used to describe the system. The second one is the semi-quantitative evaluation and involves assigning likelihood scores to the activated events and aggregating them upward through the tree. The resulting top event likelihood score is used to compare architecture variants.

The score produced by the framework is an ordinal measure of how reachable the selected top event is under the architectural conditions represented by a given variant, not an absolute probability of mission failure. This makes the output suitable for comparing design alternatives during early architecture selection, while avoiding claims of probabilistic precision that would not be justified at this stage.

Figure 2.1 presents the overall steps followed when applying the proposed framework.

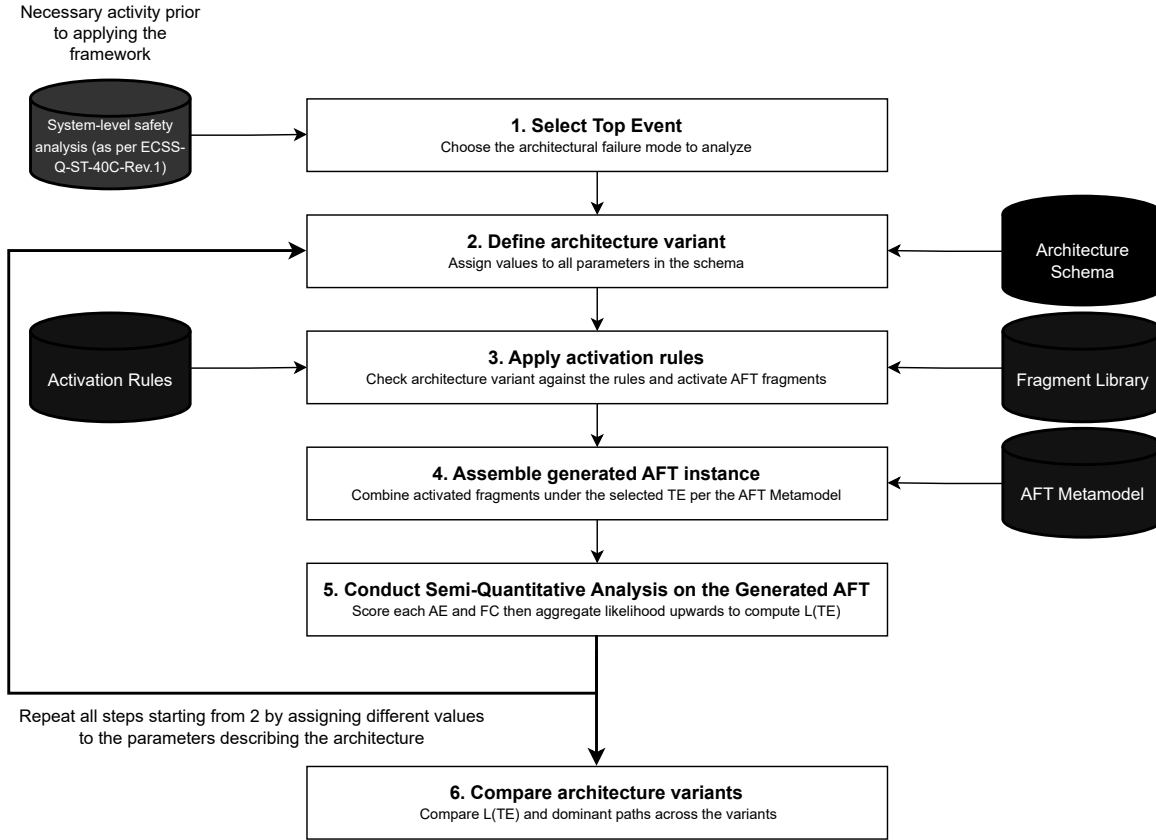


Figure 2.1: *Framework process diagram*

2.2 Scope and Modelling Assumptions

The framework is intended as a reusable method for parameter-driven AFT generation and comparison. In this thesis, the method is instantiated for multi-tenant SataaS OBSW architectural class. The parameter set, fragment library, and activation rules are therefore tailored to this architectural class. Applying the same method to another domain, for example the ground station architecture, would require redefining these artefacts according to the relevant system boundaries, interaction paths, authority relationships, and control mechanisms, without modifying the general idea and steps of the framework.

The framework is applied during the early architecture phase of a mission, when alternative designs can still be compared. At this stage, the analysis focuses on how architectural choices expose, constrain, or contain attack-fault paths towards a selected top event. The result is intended to support design comparison, not to replace detailed safety, security, or implementation verification activities.

The in-scope elements are untrusted tenant software, trusted platform software, tenant-platform interfaces, tenant-tenant communication, shared resources, isolation mechanisms, permissions, update mechanisms, and access to spacecraft-facing functions. The analysis covers paths that originate from tenant software or from its interaction with platform-controlled resources.

Ground segment compromise, physical tampering, malicious hardware and supply-chain compromise, RF-layer attacks, detailed hardware faults, and implementation-level vulnerabilities are outside the direct scope of this instantiation. They may be considered in broader mission-level analyses or included in future extensions through additional parameters, fragments, and activation rules.

The top event is assumed to be selected before AFT generation begins, based on prior system-level safety, security, or mission consequence reasoning. The framework does not determine mission severity, it analyses how reachable the selected top event becomes under a given architecture variant and supports comparison between variants under the same modelling assumptions.

Tenant software is treated as untrusted from an assurance perspective. This assumption does not automatically imply malicious intent. Tenant behaviour may become unsafe because of faults, unexpected inputs, misconfiguration, or deliberate misuse due to tenant software originating from different developers or users. Since these causes may lead to similar architectural violations, fault-induced and attack-induced conditions are represented within the same AFT structure.

A generated AFT represents structural applicability, not event occurrence. Activating a fragment means that the corresponding path is relevant for the analysed architecture variant, not that the attack, fault, weakness, or enabling condition has occurred. For example, if the architecture supports on-orbit tenant updates, the framework may include a fragment about unsafe update activation. If another variant does not allow tenant updates at all, that fragment would not appear.

Software criticality and security relevance are treated as related but distinct concerns. Software criticality is derived from the consequence of software failure in its system context and supports the definition of assurance effort. Security relevance is derived from the paths made available by the architecture. A tenant application supporting a low-criticality function may still contribute to a higher-consequence path if it can reach platform services, shared resources, other tenants, or spacecraft-facing interfaces. This separation is consistent with ECSS-Q-ST-80C, where software security is treated as a transversal activity that may apply independently of software criticality [55].

2.3 Framework Artefacts

2.3.1 System Description: Parameterised Architecture Input as the AFT Driver

Since the framework's end goal is to compare architecture variants, in order to compare them fairly, both must be described using the same dimensions. Parameterisation addresses this requirement. The analyst assigns one value to each dimension, so that alternatives can be com-

pared under a common structure.

Table 2.1: Example of parameterised architecture variants

Architecture parameter	Variant A	Variant B
Tenant-platform containment	containerized	shared process
Shared memory	disabled	enabled
Communication control	capability-based	basic policy
Resource isolation	quotas and watchdogs	none

A change in parameter value can affect the analysis structurally. For example, if shared memory is disabled in one variant but enabled in another, then shared-memory fragments become relevant in the latter, because tenant software may influence other software through mapped memory.

This change in parameter values can also have an evaluative effect, meaning that the fragment may not be modified structurally, but its likelihood score changes. For example, if communication between tenant and platform is API-based and mediated through the platform services and communication control is capability-based, then the API mediation path may still exist. However, capability-based access control makes the path less reachable than an unmediated or weakly checked API.

Using this type of input as the analysis driver is aligned with model-based threat analysis (MBTA), where a software architecture model is evaluated against formalised threat signatures to identify relevant threats [9]. The same principle is adopted here to AFT generation: architecture parameters are evaluated by activation rules to instantiate reusable attack-fault fragments.

The parameter set used in this thesis was selected according to four concerns: boundary crossing, component interaction, reachable authority, and control strength. Boundary-related parameters describe whether an event can remain local, propagate laterally, or cross into a trusted domain. Interaction-related parameters identify the routes through which components may influence each other, such as API call, message bus, shared memory, lifecycle update, and shared resource exhaustion. Authority-related parameters capture what can be done through those routes. Examples here include reading TM, modifying payload settings, sending command requests, changing configuration etc. Control-related parameters describe how strongly the architecture restricts access, persistence, or propagation.

Execution and isolation parameters define the trust boundaries of the architecture. The execution model captures the underlying system structure, such as a monolithic kernel, microkernel, separation kernel, or partitioned RTOS. Tenant-platform containment captures the isolation boundary between untrusted tenant software and trusted platform functions, such as container boundary or VM boundary. Tenant-tenant separation captures the boundary between tenant applications.

The isolation-related parameters are split into two dimensions: tenant-platform containment and tenant-tenant separation. Tenant-platform containment describes the boundary between un-

trusted tenant software and trusted platform functions. Tenant-tenant separation describes the boundary between tenant applications. The distinction is needed because escalation into platform authority and lateral interference between tenants produce different attack-fault paths.

Both dimensions are expressed through ordered containment levels, ranging from no enforced separation to process-based, container-based, virtualised, and partition-based isolation. These levels represent progressively stronger assumptions about memory separation, execution independence, and enforcement by the underlying runtime, OS, hypervisor, or separation kernel.

Figure 2.2 summarises the resulting isolation design space. Valid combinations represent architecture variants that can be analysed directly (represented through a check mark). Invalid combinations indicate inconsistent assumptions, such as strong tenant-tenant separation without a corresponding tenant-platform containment mechanism (represented through an X symbol). Advanced combinations represent nested designs, for example virtualisation combined with partitioning, which are possible but require additional architectural justification (represented through the triangle symbol).

T-P containment		none	logical only	process separated	container separated	VM separated	partition separated
	none	✓	✓	X	X	X	X
	logical only	✓	✓	X	X	X	X
	process sandboxed	✓	✓	✓	X	X	X
	containerized	✓	✓	✓	✓	X	X
	VM isolated	✓	✓	✓	✓	✓	△
	partition isolated	✓	✓	✓	✓	△	✓
T-T separation							

Figure 2.2: *Isolation design space defined by tenant-platform containment and tenant-tenant separation*

The vertical axis (T-P containment) represents the protection against untrusted software affecting the platform, while the horizontal one (T-T separation) represents protection against tenant-to-tenant interference. Consequently, moving up inside the figure increases the protection of platform from tenants, while moving right increases protection between tenant applications. From a practical interpretation point of view, each cell in the figure defines a specific execution architecture, a specific attack surface, and a specific fault propagation behaviour. Therefore, each cell drives a different AFT instance.

The main principle followed throughout the development of this matrix is that the internal separation of tenants should not exceed or contradict the containment mechanism applied between tenants and the platform, unless explicitly modelling advanced nested designs. This is the reason

why some combinations are invalid, such as having no T-P containment, while choosing VM T-T separation. This would translate to tenants being strongly isolated from each other, but not isolated from the platform, which is architecturally inconsistent: VM implies strong isolation layer, but cannot exist without also isolating from platform.

The triangle symbolizes advanced combinations, such as T-P: VM-isolated, while T-T: partition-separated. This implies that tenants are inside VMs and inside those VMs, partitions are used. This is not inherently impossible, but it is complex and uncommon.

Interaction surface parameters describe how influence can move across those boundaries. Tenant-platform communication represents the service or message path between tenant software and platform software. Tenant-tenant communication captures direct or mediated interaction between tenants. Shared memory is modelled separately because mapped memory introduces paths that differ from message-based communication. Spacecraft-facing interface access captures cases where tenant software can read from, write to, or command functions normally expected to remain under trusted platform control.

Access and mediation parameters refine the meaning of an existing path. Communication control captures whether interaction is unmediated, policy-restricted, or capability-based. Data and operation access distinguish read-only access, write access, read-write access, and command-capable access where applicable. Separate controls are used for shared memory and spacecraft-facing interfaces because these paths may bypass ordinary service mediation.

Lifecycle and integrity parameters describe whether software can be trusted during deployment, update, and startup. Platform update control and tenant deployment determine how software is accepted, replaced, or activated. Code signing, boot integrity, and key protection represent the trust mechanisms used to prevent unauthorised execution or weakened verification. These parameters matter because a compromised update path or weak trust root can enable persistent or high-authority failures.

Resource isolation is treated separately because interference may occur without direct data access or command authority. A tenant may affect platform services or other tenants by exhausting CPU time, memory, storage, or I/O bandwidth. This parameter group allows the generated AFT to include denial-of-service and shared-resource interference paths alongside direct communication and access-control paths.

The parameter space allows the representation of a wide range of architectures, from flat execution environments to high-assurance partitioned systems.

2.3.2 Attack-Fault Tree Metamodel

This framework is designed to generate AFT instances from the architectural description of the flight software, meaning that the same architecture may be analysed with respect to different top events or that the same top event may produce different trees for different architecture configurations. Without a consistent definition of the event types that may appear in a generated AFT and of the way these events are organized, the resulting trees would not be comparable across

analyses. The purpose of the AFT metamodel is to define a common structural vocabulary for all generated trees, so that differences between AFT instances can be interpreted as consequences of architectural variation rather than as differences in analyst modelling choices.

The metamodel aims to capture the level of abstraction that can reasonably be inferred from the software architecture description at the preliminary design stage. At this stage, the available information is usually sufficient to reason about architectural boundaries, communication paths, isolation mechanisms, shared resources, access-control assumptions, and possible classes of weakness. It is not necessarily sufficient to model detailed implementation behaviour, exact exploit chains, or mission-specific failure probabilities. For this reason, the metamodel is positioned between high-level system safety analysis and low-level vulnerability analysis: it represents architecture-level failure mechanisms and their enabling attack/fault conditions, while using CWE categories only as localised weakness indicators rather than as evidence of concrete vulnerabilities. The proposed metamodel imposes a fixed layered structure on every generated AFT instance, as illustrated in Figure 2.3. This is a deliberate modelling choice. Classical FTA gives the analyst considerable freedom to decompose the top event at the level of detail and with the event categories most appropriate for the system under analysis. This flexibility is valuable when the objective is to construct the most complete failure model of a single fixed system. However, it also makes comparison difficult, because two trees may differ not only because the systems differ, but also because the analysts selected different abstraction levels, event names, or decomposition strategies.

In contrast, the objective of this framework is not to produce a complete safety or security model of one finalised spacecraft design, but rather to support early comparison between alternative onboard software architecture variants for multi-tenant SataaS missions. For this purpose, comparability and repeatability are more important than unrestricted modelling freedom. Consequently, when two architecture variants generate different AFT structures, these differences can be traced back to differences in the selected architectural parameters and activated fragments.

Automation is also easier under this structure, because each event type has a predefined role in the tree and reusable AFT fragments can be written against the metamodel and activated through rules evaluated over the architecture schema. In this way, the framework separates the definition of possible failure structures from the selection of a concrete architecture variant. The fragment library defines what types of failure paths may exist within the framework scope, while the rule set determines which of those paths are structurally applicable to the analysed variant.

This restriction introduces a known limitation. Some failure paths may require a different decomposition structure or a more detailed implementation model than the one supported by the metamodel. Such paths are outside the current scope of the framework and would require either manual extension or future refinement of the fragment library and schema. Nevertheless, the restriction is appropriate for the intended use of the framework, because it makes generated AFTs reproducible, structurally comparable, and suitable for semi-quantitative comparison across architecture variants.

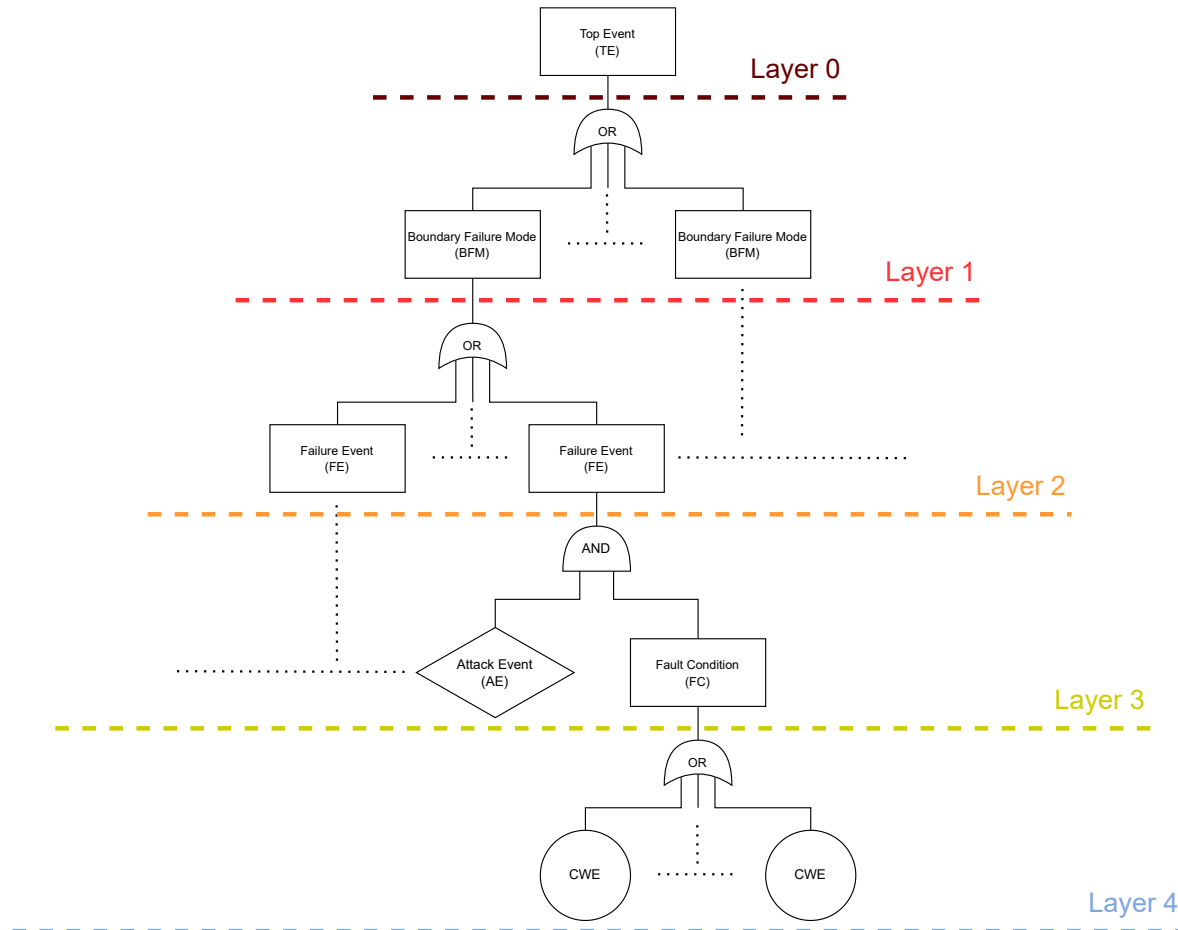


Figure 2.3: *Proposed AFT metamodel*

Layer 0: Top Event

A fault tree, and by extension an attack-fault tree, is built around a specific undesired system state: the top event (TE). The *Fault Tree Handbook* defines it as the failure mode of the system being analysed [31]. In this thesis, the TE is expressed as the failure of the analysed software architecture to preserve a required property. The choice of which property to analyse follows from a prior system-level safety, security, or mission consequence assessment, which may use methods such as FMEA, FTA, or hazard analysis to determine what functions are safety- or mission-critical and which failure states require further analysis. The framework then takes the identified failure state and translates it into an architecture-level property violation that can be decomposed through the AFT structure.

A consequence such as loss of spacecraft, loss of payload service, or loss of attitude control may involve several subsystems and failure mechanisms outside the scope of the software architecture model. The TE is therefore written as the architecture-level failure condition that can contribute to such consequences. For example, the prior safety assessment may identify that mission-critical failure states share a contributing condition: tenant software operating beyond its authorised execution context, with access to platform services that it should not be able to influence. That condition is not directly representable as a mission consequence, since it sits at the software architecture level rather than at the level of hardware subsystems or operational outcomes. The framework takes it and expresses it as a TE: *Tenant-Platform Privilege Boundary Enforcement Failure*, which describes the software-architecture condition under which a tenant application can gain platform-level privileges or influence platform services beyond its authorised scope, making downstream failure states reachable through the architecture.

In summary, TE selection follows two steps. First, the prior system-level analysis identifies an undesired failure state relevant to the mission. Second, this concern is translated into the architectural guarantee whose failure the framework can analyse. If prior analysis identifies a high-consequence failure state with a contributing condition at the software architecture level, that condition is translated into the architectural guarantee whose violation the framework can analyse and decompose.

A single AFT instance is generated per selected TE. Different TEs require separate instances, since each represents a different violated architectural guarantee, and the decomposition below the TE changes accordingly.

Layer 1: Boundary Failure Mode (BFM)

Boundary Failure Modes (BFMs) form the first decomposition layer below the selected top event. It is broader than an FE, but more specific than the TE. In the metamodel, the TE is connected to its BFMs through an OR gate: each BFM represents an alternative architectural way in which the top event may be realised. One activated BFM is therefore sufficient to contribute to the top event, although several BFMs may be present simultaneously in the same generated AFT instance. For example, if tenant software can directly access a platform command service, then the privilege

boundary may fail even if cryptographic keys remain protected.

A BFM is a broad architectural failure class: a failed boundary, an ineffective control path, a broken trust relationship, or an unmet protection obligation. It is therefore positioned between the TE and the concrete failure events (FEs) below it. For example, *Replayed platform command request accepted* is too specific for this layer, because it describes a particular failure path. A broader failure mode such as *tenant-platform communication mediation failure* is more appropriate, because it captures the architectural class under which replay, forgery, missing authorisation, or confused-deputy behaviour can be represented as lower-level FEs (subsection 2.3.2).

The candidate BFMs are defined in the AFT fragment library (subsection 2.3.3), which is developed independently of any concrete architecture variant. For a selected TE, the fragment library specifies the possible first-level architectural failure modes through which that top event may occur. They are defined once as reusable decomposition elements and are later selected during AFT generation, based on the activation rules (subsection 2.3.4). A BFM is included in the generated AFT only if the selected variant activates that failure mode or at least one FE belonging to it. In this way, the fragment library remains variant-independent, while the generated AFT instance remains variant-specific.

The derivation of BFMs follows a repeatable procedure. First, the selected TE is rewritten as a violated architectural guarantee. Second, the architecture schema is inspected to identify the parameter groups that define, enforce, or influence that guarantee. Third, each relevant parameter group is translated into a required architectural protection, expressed as a positive statement of what the architecture must preserve. Fourth, this protection statement is negated to obtain a candidate BFM. Finally, the candidate is checked against the BFM acceptance criteria and, if accepted, is decomposed into FEs at the next layer of the AFT.

1. **State the architectural guarantee.** Define what property the architecture is expected to preserve in order to avoid the selected TE.
2. **Identify relevant schema dimensions.** Select the architecture-schema parameters that define or influence this guarantee.
3. **Group related parameters.** Combine parameters that enforce the same type of boundary, control path, trust relation, or protection mechanism.
4. **Formulate the required protection.** Express what the architecture must ensure for each parameter group.
5. **Negate the required protection.** Convert each required protection into an architectural failure mode. This failure mode becomes a candidate BFM.
6. **Apply acceptance criteria.** Keep the candidate only if it is related to the TE, traceable to the schema, variant-independent, architectural in scope, and decomposable into concrete FEs.

For the top event *Tenant-Platform Privilege Boundary Enforcement Failure*, the architectural guarantee is that tenant-originated behaviour must not obtain, exercise, modify, bypass, or corrupt

authority reserved for trusted platform software. The relevant schema dimensions are those that define how the tenant-platform boundary is created, controlled, and protected: tenant-platform containment, tenant-platform communication, access-control policy, lifecycle control, code signing, boot integrity, cryptographic key protection, resource isolation, shared-state control, and tenant-tenant interaction paths that may indirectly affect platform-facing behaviour.

Each of these schema dimensions implies a required protection:

- tenant-platform containment requires tenant execution to remain separated from platform authority;
- tenant-platform communication requires tenant requests to be mediated before they reach platform services;
- access-control and identity-binding mechanisms require platform authority to be applied only to the correct tenant identity, request origin, and permission scope;
- lifecycle controls require platform-relevant software, updates, boot state, and deployment artefacts to remain trusted;
- cryptographic protection requires platform keys, credentials, tokens, and capabilities to remain inaccessible or unusable from tenant context;
- resource-isolation mechanisms require tenant resource usage not to interfere with platform mediation or boundary enforcement;
- tenant-tenant separation requires that one tenant cannot use another tenant as an indirect path toward platform authority.

Negating these required protections produces the candidate BFM for the selected TE. For example, the requirement that tenant execution must remain separated from platform authority becomes *execution containment failure*. The requirement that tenant requests must be mediated before reaching platform services becomes *tenant-platform communication mediation failure*. The requirement that platform authority must be bound to the correct tenant identity and scope becomes *platform authority or tenant-context binding failure*. The requirement that platform trust material must remain inaccessible from tenant context becomes *platform trust material exposure or misuse*.

Not every possible failure statement is accepted as a BFM. A candidate must satisfy five criteria. First, it must be directly related to the selected TE. Second, it must be traceable to one or more architecture-schema parameters. Third, it must be independent of a single architecture variant. Fourth, it must describe an architectural failure mode rather than a specific exploit technique or CWE. Fifth, it must be decomposable into concrete FEs at the next layer of the AFT.

These criteria keep the BFM layer at the correct level of abstraction. A statement such as "the architecture is insecure" is too broad, because it cannot be tied to a specific architectural protection. Similarly, a statement such as "a replayed command request is accepted" is too narrow, because it

already describes a concrete FE. The BFM layer therefore acts as the bridge between the selected TE and the reusable lower-level fragments.

The resulting BFM set is considered complete only within the scope of the current architecture schema and the selected TE. It is not claimed to cover every possible SataaS implementation. If the schema is extended with additional architectural properties, the BFM set and the associated fragment library must be reviewed and extended accordingly.

Layer 2: Failure Event (FE)

Failure Events (FEs) form the second decomposition layer in the metamodel. They refine each BFM into architecture-level events that explain how the broader boundary failure may occur. A BFM names the failed protection class; an FE identifies a specific path within that class.

A BFM is connected to its FEs through an OR gate, since any one FE is sufficient to realise the parent failure mode. For example, under *tenant-platform communication mediation failure*, possible FEs include *unauthorised platform request accepted*, *replayed platform command request accepted*, or *tenant request executed under incorrect platform authority*. These events are narrower than the BFM, while still remaining above implementation bugs, exploit steps, or individual vulnerability identifiers.

The FE layer is where architecture parameters start to produce visible differences between generated trees. A fragment containing an FE is included only when the analysed variant provides the required architectural conditions. Shared-memory interference, for instance, requires a relevant shared-memory path. Misuse of a spacecraft-facing service requires interface reachability and an access level that can affect that service.

Each FE is decomposed into an Attack Event (AE) and an enabling Fault Condition (FC). The AE captures the tenant-originated action, interaction, or misuse attempt that exercises the path. The FC captures the missing control, architectural weakness, misconfiguration, or fault condition that allows the AE to produce the FE. Both must be present for the FE to occur, which is why this layer uses an AND decomposition.

This structure supports the safety-security interpretation of the AFT. The AE may represent deliberate misuse, faulty behaviour, unexpected input, or an invalid interaction. The FC explains why the architecture fails to contain or reject that behaviour. Together, they describe how tenant-originated behaviour becomes an FE at architecture level.

An FE is included in the fragment library only if it satisfies four conditions. It must refine one BFM, remain at architecture level, be traceable to one or more architecture parameters, and decompose into an AE and an FC. These criteria keep the layer usable for automated generation: detailed enough to activate and score, but general enough to reuse across variants.

The FE layer is therefore the main bridge between architectural failure modes and lower-level causes. Differences in the FE set of two generated AFTs indicate that the analysed variants expose different paths towards the same BFM or selected top event.

Layer 3: Attack Event (AE) & Fault Condition (FC) Pair

Layer 3 explains why an FE can occur. Each FE is decomposed into an AE-FC pair. The AE represents the tenant-originating behaviour that exercises a path, while the FC represents the architectural condition that allows this action to become effective.

The AE is defined as an adversarial or misusing action that stems from the untrusted software. Since tenants may originate from different third-parties and may not be developed, verified, or assured to the same level as trusted platform software, an attack event may describe a crafted or invalid request, replay, spoofing, injection, abuse of an exposed interface, misuse of delegated authority, or another interaction that challenges an intended boundary, interface, or authority path.

The FC captures the missing or weakened protection that allows the AE to produce the parent FE. Examples include absent mediation, incorrect authority binding, weak isolation, insufficient resource control, exposed trust material, or an unsafe update condition. The FC is the architectural enabler of the FE. Without it, the AE may still occur, but it should be rejected, contained, or prevented from propagating. This pairing avoids modelling tenant behaviour and architectural weakness as independent causes of the FE. In most cases relevant to this framework, an invalid tenant-originated action becomes harmful only when the architecture permits it to cross a boundary, reach a protected function, or persist in a trusted context.

At the conceptual level, AEs and FCs may be reused across several FE branches. The same AE may be associated with different FCs when the same unintended action is enabled by different weaknesses. For example, a crafted platform API request may become relevant because of weak authorisation, incomplete input validation, or fail-open mediation. Similarly, the same FC may be associated with different AEs when one architectural weakness enables several unintended actions: weak command authorisation may enable an unauthorised command request, a forged command request, or abuse of delegated platform authority.

In the current metamodel, these concepts are represented as AE-FC pairs under each FE branch, keeping the generated structure tree-oriented: each FE branch contains one associated AE and one associated FC. If the same AE or FC concept is relevant in more than one place, it is repeated as a local instance in the corresponding FE branches. A more general DAG-based model could represent repeated concepts as shared nodes. This framework keeps a tree structure to preserve readability, direct traceability, and straightforward likelihood propagation.

The AE-FC pair also supports semi-quantitative evaluation. The AE represents the accessibility of the action or interaction, while the FC represents the weakness of the architectural control. Their combined score expresses the likelihood of the FE under the analysed architecture variant. This keeps scoring tied to the same reasoning used for generation: paths become more relevant when unintended tenant behaviour is easy to exercise and the corresponding control is weak or absent.

Layer 4: Localised CWEs as Basic Events

Layer 4 contains the basic events in the metamodel, which are represented through localised CWE entries. The term *localised* suggests that a generic CWE entry is interpreted in relation to the architectural element affected by the parent FC. For example, an authentication-related CWE is meaningful only when it is attached to a specific mediation or access-control condition in the architecture.

CWEs are used as basic events because they provide a standardised vocabulary for software weaknesses, while remaining abstract enough for early architecture analysis. They also support traceability to recognised security taxonomies, which makes the fragment library easier to review and extend. Within this metamodel, a CWE is treated as atomic: it is not decomposed further by the framework, even though the same weakness could be analysed in more detail in a lower-level security assessment.

CWE selection follows five rules. First, the weakness must be software-relevant and applicable to software systems, execution environments, interfaces, or access-control logic. Hardware-only, physical, or organisational weaknesses are outside the scope of this layer. Second, the weakness must be architecture-relevant: it must plausibly affect a boundary, communication path, control mechanism, shared resource, lifecycle mechanism, or spacecraft-facing interface represented in the framework. Third, the CWE should be platform-independent enough to remain reusable across architecture variants: entries tied to a specific product, library, or implementation technology are avoided unless the fragment is explicitly intended for that environment. Fourth, the abstraction level must match the limit of resolution of the basic events in the metamodel. Base-level CWEs are preferred because they usually describe identifiable weakness mechanisms, while pillar-level CWEs are too broad and variant-level CWEs often introduce implementation detail that is not available during preliminary design. Class-level CWEs may be used when they are sufficiently specific and no more suitable base-level entry is available. Fifth, the CWE must support the parent FC: a weakness is accepted only when it can help explain why the architectural control represented by the FC may fail. This rule prevents unrelated CWEs from being attached to the tree simply because they are security-relevant in a general sense. Each CWE must therefore be traceable upward to one FC, one FE, and ultimately to the selected BFM and TE.

2.3.3 AFT Fragment Library

The AFT fragment library is the artefact that defines the reusable failure paths activated during the AFT generation, containing partial AFT structures written according to the metamodel introduced in Subsection 2.3.2. Each fragment represents one architecture-level path from a BFM to its lower-level causes. This library defines what can appear in a generated AFT before any architecture variant is selected.

A fragment is linked to a selected top event and to one parent BFM. Within that BFM, it specifies an FE, the associated AE and FC pair, and the CWEs that may support the FC, keeping every fragment consistent with the metamodel. A fragment is smaller than a full AFT, yet large

enough to express a complete reusable path.

The library is developed independently of individual architecture variants. Its content is derived with respect to the selected top event and to the parameter groups that may affect those guarantees. For example, the *Tenant-Platform Privilege Boundary Enforcement Failure* top event leads to fragments involving containment failure, communication mediation failure, authority binding failure, lifecycle integrity failure, but not to fragments whose effect is limited to tenant-tenant interference, spacecraft-facing command validation, or general software execution integrity, unless those paths directly enable tenant influence over trusted platform authority.

Each fragment has to satisfy four conditions before being included in the library. First, it must refine a BFM that belongs to the selected top-event decomposition. Second, it must describe an architecture-level attack-fault path, without depending on a specific software product, version, or implementation bug. Third, it must be traceable to one or more architecture parameters, so that activation can be evaluated systematically. Fourth, it must be decomposable into the FE, AE, FC, and CWE structure required by the metamodel.

AFT fragments, or sub-trees, support reusability and comparison. The same fragment can be evaluated against multiple architecture variants, and differences between generated trees can be traced to the parameters that activate or suppress fragments. This avoids rebuilding each AFT manually and keeps the comparison focused on architectural differences.

The current library is developed for the top event *Tenant-Platform Privilege Boundary Enforcement Failure*. Other top events require additional fragment sets, because each top event represents a different violated architectural guarantee. The metamodel remains the same, but the BFMs, FEs, AE-FC pairs, CWE mappings, and activation rules have to be defined for the new top-event scope.

2.3.4 Activation Rules for Fragment Instantiation

The framework uses a set of activation rules to determine which fragments are structurally applicable to the selected architecture. Activation means that the architecture variant contains the relevant path, boundary, access relation, or weak enforcement condition for the corresponding fragment to be included in the generated AFT.

The fragment library and activation rules are related, but they have different roles. The fragment library is defined first from the selected top event and the AFT metamodel, because it specifies the possible paths that may appear in the tree. The activation rules are then constructed by mapping each fragment to the architecture parameters that make it structurally applicable. The process is iterative in implementation, since rule development can reveal where a fragment needs to be refined or where the schema lacks a parameter needed for more precise activation.

The rule logic follows the layer structure of the AFT metamodel: BFMs are activated by the existence of relevant architectural paths or protection mechanisms and FEs are activated when the architecture makes a specific failure outcome structurally possible. FCs are activated when the selected architecture variant contains, exposes, or cannot rule out the weak, missing, insufficient,

or structurally relevant control condition that enables the corresponding FE. At the early architecture stage, FC activation means that the enabling condition is relevant to the analysed architecture path, not that an implementation-level weakness has already been proven. AEs and CWEs are included structurally through their activated parent nodes. An AE is included as the tenant action associated with an activated FE, while CWEs are included as weakness classifications associated with an activated FC.

The rules are developed manually from the relationship between the selected top event, the fragment library, and the architecture schema. First, the selected TE is decomposed into BFMs and FEs according to the AFT metamodel. Each BFM is associated with a broad architectural concern, such as containment, communication mediation, lifecycle integrity, shared memory, or resource mediation. Each FE is then checked against the architecture schema to identify which parameters make that failure outcome structurally possible. Finally, the FC associated with the FE is mapped to the parameter value, or combination of values, that represents the weak, missing, or insufficient control condition.

In this sense, a rule is the explicit encoding of the question: under which architecture conditions is this fragment relevant? For example, a shared-memory fragment is only relevant if the schema indicates that shared memory exists in a tenant-platform or tenant-tenant scope. A command-service fragment is only relevant if tenant-platform communication exists and the tenant has command-capable access. A lifecycle-integrity fragment is only relevant if update, deployment, signing, boot, or key-protection mechanisms expose a path by which untrusted software or weakened verification can affect trusted platform execution.

The construction of a rule therefore follows the same pattern for each fragment. The analyst identifies the architectural precondition for the BFM, the more specific architectural condition for the FE, and the enabling weakness or control condition for the FC. These conditions are then expressed using the parameter names and allowed values from the architecture schema. Rules may use a single parameter value, such as `TP_CONTAINMENT = containerized`, or a combination of values, such as `SHM_SCOPE` indicating tenant-platform sharing together with `SHM_ACCESS` indicating write-capable access.

When the schema does not contain a parameter fine-grained enough to confirm a specific implementation weakness, the FC may inherit the FE condition. In that case, the rule does not claim that the weakness has been proven; it only states that the architecture exposes the path in which that weakness would be relevant, leaving detailed implementation assessment to later analysis.

Formally, the activation approach can be summarized as seen in the table 2.2:

Table 2.2: Activation Logic of Event Types

Node Level	Activation Logic	Example
TE	Selected before generation, not activated by rules	N/A
BFM	Activated when the relevant architectural path/mechanism exists	If TP_COMM != none, then T-P communication failure mode is relevant
FE	Activated when a specific failure is possible within the active BFM	If TP_DATA_ACCESS = command, then unauthorized command acceptance is a possible outcome
FC	Activated when the specific weakness that enables FE exists	If TP_COMM_CONTROL = unmediated, then missing authorization is the enabling weakness
AE	Included structurally with activated parent FE	If TP_COMM != none AND TP_DATA_ACCESS = command, then activate FE: Unauthorized platform command accepted, which includes AE: Tenant sends unauthorized platform command
CWE	Included structurally with activated parent FC to classify the type of weakness	If TP_COMM_CONTROL = unmediated, then activate FC: Command authorization not enforced on all paths, which includes CWE-862 (Missing Authorization)

This results in a progressively more specific activation process down the AFT hierarchy. For example, if the architecture variant defines a tenant-platform communication path, the corresponding communication-control BFM is activated. If that path provides command-level access, then the FE representing unauthorized platform command acceptance becomes structurally relevant. If the communication control is unmediated, the FC representing missing or insufficient authorization is activated. The associated AE, such as a tenant submitting an unauthorized platform command, is included as the attack-side child of the FE. The CWE nodes associated with the FC, such as missing authorization or improper access control classifications, are then included under the FC.

This distinction between direct activation and structural inclusion avoids unnecessary duplication in the rule base. Architecture parameters directly determine the applicability of BFMs, FEs, and FCs. AEs and CWEs are not independently activated by architecture parameters because they are semantically tied to their parent nodes. The AE explains what the tenant does in the activated failure path, while the CWE explains the weakness class associated with the activated fault condition.

The activation rules therefore provide the link between the architecture schema and the AFT fragment library. The architecture schema defines the allowed architectural features and control mechanisms; the fragment library defines reusable AFT structures; and the activation rules determine which fragments are instantiated for a particular architecture variant. The generated AFT is thus variant-specific while still conforming to the generic metamodel.

2.3.5 Attack-Fault Tree Generation

AFT generation is the step that combines the selected top event, the parameterised architecture input, the fragment library, and the activation rules into one generated tree. The process is deterministic: given the same top event, architecture variant, fragment library, and rule set, the same AFT structure is produced.

Generation starts with selecting the TE. The corresponding fragment set is retrieved from the library, including the BFM, FEs, AE-FC pairs, and CWEs. The architecture variant is then evaluated against the activation rules, with each rule checking whether the architectural conditions required by a fragment are present, such as a communication route, shared-memory path, direct interface access, weak mediation, insufficient containment, or exposed lifecycle control.

Fragments whose activation conditions are satisfied are instantiated into the generated AFT. Inactive fragments are excluded, so the resulting tree contains only paths that are structurally meaningful for the analysed variant. Activation therefore indicates applicability, not occurrence. A generated branch shows that the architecture exposes a path that can contribute to the selected top event under the assumptions of the framework; it does not claim that the corresponding attack, fault, or weakness has happened.

The instantiated fragments are assembled according to the metamodel. The selected TE forms the root of the tree. Activated BFMs are connected below it as alternative architectural failure modes. Each BFM is decomposed into its activated FEs, and each FE is decomposed into the AE-FC pair that explains the path. Localised CWE entries form the Basic Event layer below the relevant FCs. The gate structure is inherited from the metamodel: OR gates represent alternative ways of reaching the same parent event, while the AE-FC pair uses an AND relation because both the triggering behaviour and the enabling condition are required for the FE.

The generated AFT also preserves traceability information. Each activated fragment remains linked to the architecture parameters and rules that caused its inclusion. This makes it possible to explain why a branch appears in one architecture variant and disappears in another. It also separates structural generation from semi-quantitative evaluation: the generation step decides which paths are present, while the scoring step described in Section 2.4 evaluates their relative likelihood.

When several architecture variants are analysed, the same generation procedure is applied independently to each variant. The resulting AFTs can then be compared structurally by examining which BFMs, FEs, and lower-level paths are shared, added, removed, or changed between variants.

2.4 Semi-Quantitative Analysis of the Generated AFTs

The purpose of the semi-quantitative analysis stage is to evaluate the AFT paths generated for each architecture variant. The generated tree identifies which failure paths are structurally applicable to a selected architecture, but the presence of more branches does not automatically

indicate a less secure variant. Therefore, AFT generation and AFT quantitative evaluation are treated as two separate activities.

The distinction follows the general logic of FTA: a fault tree is first a qualitative model of the logical relationship between events, and it can then be evaluated quantitatively if suitable probability data are available. The Fault Tree Handbook distinguishes qualitative evaluation, based on the structure of the tree and minimal cut sets, from quantitative evaluation, which uses probability information associated with the events [31]. In this framework, exact probability data is unavailable at the architecture design stage, so a semi-quantitative likelihood scoring method is used instead. The generated AFT is first used as a structural model of applicable attack-fault paths, and likelihood scores are then assigned and propagated upward through the tree to produce a top event likelihood score for each architecture variant. This score is the primary output of the analysis and is used to compare how different architectural choices affect the reachability of the selected top event.

Separating fragment activation from likelihood scoring is important especially in cases where an architecture with more security mechanisms activates more AFT branches. For example, code signing, boot integrity, platform APIs, capability mechanisms, and key protection reduce certain risks, but they also introduce enforcement points that can fail, be misconfigured, or be misused [56]. An architecture variant that activates more branches is not necessarily less secure—what matters is the likelihood scores of the activated paths and how they propagate to the top event likelihood.

Likelihood Definition

Each FE is modeled as the joint occurrence of an AE and an enabling FC. The AE represents what the tenant does, while the FC represents the architectural, design, or configuration weakness that makes the action effective. Example: if the FE is stated as "Replayed platform request accepted", then its decomposition would be into (1) AE: "Tenant replays an old platform request" and (2) FC: "Request freshness protection does not prevent replay acceptance". The likelihood of an FE is therefore not simply the likelihood that a tenant performs an action. The latter must also be effective under the architectural conditions represented by the FC. In the example above, replaying a platform request only contributes to the FE if the platform does not correctly enforce freshness checks, such as timestamps, sequence numbers, tenant identity binding, etc.

Since AEs and FCs are not necessarily independent, the framework follows the general probability interpretation used in FTA: the joint occurrence of two dependent events may be represented using conditional probability [31]. Thus, for an FE composed of an AE and an enabling FC:

$$P(FE) = P(AE \cap FC) \quad (2.1)$$

which may further be expressed as:

$$P(FE) = P(AE) \times P(FC|AE) \quad (2.2)$$

In words, the probability of the FE equals the probability that the AE occurs multiplied by the probability that the enabling FC is present or effective given that the AE has occurred.

Semi-Quantitative Likelihood Scoring

Exact probabilistic data is usually unavailable at the architecture-design stage because the framework operates on abstract architectural parameters rather than on a fully implemented and operational system. During this phase, details such as final implementation quality, configuration, vulnerability occurrence, attacker behaviour, and control effectiveness are not yet known with sufficient confidence. For this reason, the framework uses a semi-quantitative five-level likelihood scale for prioritisation, while retaining the probabilistic formulation as the theoretical basis for how events would be evaluated if sufficient data were available.

The likelihood score is assigned separately to the AE and the FC, because they describe different aspects of the FE. The AE score represents the feasibility of the tenant action, while the FC score represents the likelihood or effectiveness of the architectural weakness that allows the action to succeed.

The AE likelihood score evaluates how feasible it is for the tenant-originating action to be exercised through the analysed architecture. The same AE may result from deliberate misuse, compromised tenant software, incorrect integration, unsafe input generation, or incomplete assurance of a third-party tenant application. The score therefore reflects accessibility, required knowledge, timing constraints, and technical difficulty of the action, rather than the motivation of the tenant or developer.

Table 2.3: Attack Event Likelihood Scoring

Score	AE Likelihood Level	Interpretation
1	Very Low	The tenant action is very difficult and requires exceptional knowledge, rare timing, or prior compromise of another protection layer.
2	Low	The tenant action is difficult and requires specific knowledge, access to non-obvious data, or precise timing.
3	Medium	The tenant action is plausible because the tenant has access to a relevant interface, but it still requires some preparation or knowledge.
4	High	The tenant action is easy to attempt because the relevant interface or resource is directly reachable.
5	Very High	The tenant action is trivial or directly enabled by normal tenant access.

A high AE score should not be interpreted as proof of malicious behaviour. It only means that the action represented by the AE is easy to exercise from the tenant context. For example, sending a malformed request to a reachable platform API may receive a high score, while exploiting a VM or hypervisor boundary may receive a low score because it requires a more advanced exploit and stronger attacker capability. Whether the behaviour is malicious, compromised, accidental, or caused by insufficiently assured third-party software is outside the scoring scale. The framework treats it as attack-like because it is an active tenant-originated stimulus that may challenge an architectural boundary or authority path.

The FC likelihood score evaluates the likelihood that the relevant architectural control fails to prevent the attack event from succeeding, which is equivalent to the likelihood that the fault condition is present and effective under the selected architecture. It describes whether the relevant protection is absent, weak, optional, software-only, configuration-dependent, incorrectly bound to identity or context, or difficult to verify.

Table 2.4: Fault Condition Likelihood Scoring

Score	FC Likelihood Level	Interpretation
1	Very Low	Strong control exists and the weakness is unlikely under the selected architecture.
2	Low	Control exists and is generally strong, but residual failure is still possible.
3	Medium	Control exists, but it's configuration- dependent, complex, or only partially specified.
4	High	Control is weak, optional, incomplete, software-only, or inconsistently enforced.
5	Very High	Control is absent, unmediated, or directly permits the weakness.

This scale assumes that score 1-4 refer to cases where some form of control exists but with different levels of strength or assurance. Score 5 is used when the relevant control is absent or when the architecture directly exposes the weakness and it is particularly important when AFT fragments activate precisely because no control was specified.

For example, if no replay protection is implemented for platform requests, the FC *Request freshness protection does not prevent replay acceptance* may receive a score of 5. If nonce-based replay protection exists but is not cryptographically bound to the tenant identity or request content, the FC score may be reduced to 2 to 3, depending on the strength of the implementation.

The probability formulation introduced earlier expresses the FE as the joint occurrence of the AE and FC. For the implemented scoring method, this is approximated using the product of the AE and FC likelihood scores:

$$L_{FE,raw} = L_{AE} \times L_{FC} \quad (2.3)$$

This multiplicative combination is not equivalent to the probabilistic formula of FE, since ordinal

scores are not probabilities and multiplication does not carry the same formal meaning. However, it is used because the product is monotonic in both arguments: if L_{AE} increases while L_{FC} stays fixed, L_{FE} increases and if L_{AE} decreases while L_{FC} stays fixed, L_{FE} decreases. The same holds symmetrically for L_{FC} —an increase in either input cannot decrease the resulting FE score. This is the property that makes the rule consistent with the underlying causal logic of the framework: a more feasible tenant action increases L_{FE} only to the degree that the architectural weakness allows it to succeed, and a weaker architectural control increases L_{FE} only to the degree that the tenant action is actually attempted. Neither dimension alone determines the outcome, both must be present for the FE to be realized, and the formula reflects this by ensuring that a very low score on either argument suppresses the overall L_{FE} , regardless of how high the other argument scores. The scores and their products are therefore interpreted as relative ranking indices rather than probability estimates, and are used exclusively for comparison between architecture variants evaluated under the same framework.

The raw product ranges from 1 to 25 and is then mapped back to the five-level likelihood scale:

Table 2.5: Failure Event Likelihood Score Derivation

$L_{FE,raw}$	L_{FE}	Likelihood Level
1-2	1	Very Low
3-6	2	Low
7-12	3	Medium
13-19	4	High
20-25	5	Very High

This mapping preserves a five-level scale while still allowing differentiation between combinations where an easy attack event should not automatically imply high FE likelihood if the corresponding fault condition is strongly controlled. Similarly, a highly enabling fault condition does not automatically produce a high FE likelihood if the attack event is very difficult.

Likelihood Propagation

The likelihood scores defined so far apply to individual FEs at Layer 3 of the AFT. To compute a likelihood for an entire activated fragment and ultimately for the top event, the FE likelihood scores are propagated upward through the gates of the tree. The metamodel in Figure 2.3 introduces AND logic only within the FE decomposition into an AE and an enabling FC. In the proposed semi-quantitative method, this AND relationship has already been captured through the multiplicative combination of the AE and FC scores.

Above the FE level, the generated AFT uses OR gates to represent alternative ways in which a parent event may occur. In classical FTA, the OR gate represents the Boolean union of its input events: any one or more of the input events is sufficient to cause the event above the gate [31]. The Fault Tree Handbook expresses this for two events as $Q = A + B$ and gives the probabilistic

form as:

$$P(Q) = P(A) + P(B) - P(A \cap B) \quad (2.4)$$

This formula only applies when actual event probabilities are available. So, for ordinal likelihood scores, the maximum operation is adopted as the ordinal aggregation rule for OR gates:

$$L_{OR} = \max(L_{input_1}, L_{input_2}, \dots, L_{input_n}) \quad (2.5)$$

This rule preserves the qualitative meaning of the OR gate: since any child event is sufficient to produce the parent event, the parent likelihood should be at least as high as the most likely child path. In other words, the likelihood of the parent event is determined by the most accessible or most plausible activated input. The resulting value is interpreted as a prioritisation score, not as the exact probability of the parent event occurring.

The CWEs at Layer 4 do not participate in likelihood propagation: likelihood is scored at the FC level, and the CWE decomposition serves a structural and traceability role rather than introducing additional likelihood information. Propagation therefore begins at the FE layer, where L_{FE} has been computed for each activated FE, and proceeds upward through successive OR gates to compute the likelihood of each BFM and finally of the TE:

$$L_{BFM} = \max(L_{FE_1}, L_{FE_2}, \dots, L_{FE_n}) \quad (2.6)$$

$$L_{TE} = \max(L_{BFM_1}, L_{BFM_2}, \dots, L_{BFM_m}) \quad (2.7)$$

The TE likelihood is therefore determined by the most accessible activated FE in the most accessible BFM, reflecting the adversarial assumption that a motivated attacker exploits the weakest available path.

2.4.1 AFT Comparison Across Architecture Variants

The comparison of architecture variants is the final output of the framework and draws on both the structural and quantitative results produced in the preceding subsections. Structural comparison examines which fragments are activated in each variant and how the tree instances differ. Quantitative comparison examines the top event likelihood scores produced by the semi-quantitative analysis. Together these two dimensions provide a complete picture of how different architectural choices affect the reachability of the selected top event.

Structural Comparison

The structural comparison examines the generated AFT instances for each architecture variant and identifies which BFMs, FEs, and paths are activated, suppressed, or shared across vari-

ants. Two variants generated for the same top event will share the same metamodel structure but differ in which fragments are activated based on their parameter values. A fragment that is activated in one variant but not in another reflects a genuine architectural difference: the presence or absence of a specific attack surface, communication path, or control mechanism.

Structural comparison alone does not determine which variant is more secure. A variant that activates more fragments is not automatically less secure than one that activates fewer. A variant with stronger security mechanisms may activate more branches precisely because those mechanisms introduce enforcement points that can fail, be misconfigured, or be misused. The structural comparison therefore serves to identify what is different between variants and why, not to rank them directly. The structural differences that matter most are those that introduce new BFMs or FEs that were not present in the other variant, since these represent qualitatively new attack-fault paths that exist in one architecture but not the other. Differences that only affect the number of FEs within an already-activated BFM are less significant structurally, since the BFM was already reachable.

Semi-Quantitative Comparison

The semi-quantitative comparison ranks variants by their top event likelihood score L_{TE} . The variant with the lower L_{TE} is less exposed to the selected top event under the given mission context—it is structurally harder for an attacker to reach the top event through the activated paths in that configuration.

Since L_{TE} is derived from the same scoring method and propagation rules for all variants, differences in L_{TE} can be interpreted under the same modelling assumptions, rather than analyst subjectivity. A variant whose parameters produce stronger FC scores (meaning its architectural controls are better specified and more consistently enforced) will produce lower L_{FE} values, which propagate to a lower L_{BFM} and ultimately a lower L_{TE} .

The fragment-level scores are a secondary diagnostic output that explains what drives L_{TE} . The dominant path is the activated FE with the highest L_{FE} that propagates through the dominant BFM to determine L_{TE} . Identifying the dominant path tells the analyst where the architectural exposure is concentrated and which specific control, if strengthened, would most reduce L_{TE} . This diagnostic use of path scores is especially useful when two variants have similar L_{TE} values, but the dominant paths may differ between them, revealing different architectural weaknesses even when the overall exposure is comparable.

Combining Both Dimensions

The structural and quantitative comparisons are most informative when read together. Four cases are worth distinguishing:

1. The first case is where one variant activates fewer paths and has a lower L_{TE} . This variant is unambiguously preferable, as it exposes fewer attack-fault paths and the paths it does expose

are less accessible.

2. The second case is where one variant activates fewer paths but has a higher L_{TE} . This can occur when a simpler architecture lacks controls that, while introducing new enforcement-point failure paths, also suppress more dangerous existing paths. The variant with fewer paths may have those paths concentrated in highly accessible FEs, while the variant with more paths may distribute the likelihood across many lower-scoring FEs. The semi-quantitative score is the decisive factor here.
3. The third case is where one variant activates more paths but has a lower L_{TE} . This is the case where added security mechanisms introduce new branches but the controls they introduce are strong enough to suppress the dominant paths. The variant with more branches is the more secure one, and the structural comparison alone would have suggested the opposite conclusion. This case illustrates why structural and quantitative comparison must be used together.
4. The fourth case is where both variants have the same L_{TE} . In this case, they should be compared through the underlying likelihood profile, including the number of BFM and FEs reaching High or Very High, the dominant paths, and the distribution of scores across the tree.

Chapter 3

Framework Demonstration: Tenant-Platform Privilege Boundary Enforcement Failure Top Event

3.1 Demonstration Overview

The proof-of-concept implementation demonstrates how architecture parameters can be used as controllable inputs for AFT generation. Two architecture variants are created by assigning different values to the same parameter set, allowing the effect of architectural changes to be observed directly at the BFM, FE and FC layers of the generated trees.

The valuable insights for the analyst come from comparing the two AFTs generated from the two different architecture variants. Since structural comparison only reveals one side of potentially exploitable weaknesses, semi-quantitative analysis should also be implemented and applied during the demonstration.

The top event of choice is *Tenant-Platform Privilege Boundary Enforcement Failure* because it directly tackles the main problem addressed in the thesis: untrusted tenant software may affect trusted platform software.

This demonstration provides a basis for future formalisation and extension of reusable safety-security analysis for multi-tenant space missions during architecture design phase. By keeping the parameter schema, fragment library, activation rules, and generated AFT outputs separate and traceable, the framework supports a repeatable process that can later be extended to additional variants, top events, and mission contexts.

3.2 Encoded Framework Artefacts

The demonstration uses the artefacts defined in Chapter 2 by first defining them manually and then encoding them in machine-readable files to enable automated generation. The encoded files are divided into two types. The first type consists of data artefacts stored in the `data` directory, which are YAML files that define the architecture schema, architecture variants, fragment library, activation rules, and the generated AFT outputs. These files contain the modelling content used by the framework. The second type consists of implementation support files stored in the `src` directory, including the Python scripts used for validation, generation, visualisation, scoring, and comparison.

YAML was selected as the file format for the modelling artefacts because it provides a readable and structured way to encode hierarchical information. Nested relationships appear in all main files: between parameters and their values, in the layer decomposition (from BFM to CWEs), and in activation-rule conditions. The same information could be encoded in other machine-readable formats, but the current choice makes the artefacts easier to inspect, edit, and review during an early-stage research implementation.

Another reason for using YAML is that it fits the prototype implementation workflow. YAML files are easy to parse in Python, can be stored under version control, and allow comments to be included in template files. This is useful for documenting allowed values, modelling assumptions, and consistency constraints close to the artefact being edited. The Python scripts process the encoded artefacts, validate them, generate AFT outputs, and produce visualisation, comparison, and likelihood-analysis reports. The scripts therefore support the workflow, while the YAML files remain the primary location of the modelling assumptions and framework inputs.

3.2.1 `architecture_schema.yaml`

This file contains all the design dimensions used to describe each architecture variant. There are 20 parameters in total: 3 in the first category, 4 in the second, and 13 in the last one. This number was selected as a prototype-level balance between coverage and manageability. The parameter set is large enough to cover the architectural properties most relevant to tenant-platform attack-fault propagation, but it remains small enough to keep the prototype understandable, traceable, and feasible within the scope of this thesis. Adding more parameters at this stage would increase the number of possible variant combinations, activation rules, and scoring cases, making the demonstration more difficult to validate without necessarily improving the clarity of the framework.

The same schema is used for both evaluated variants; the difference appears in the assigned value, not in the structure of the input file.

Table 3.1: Architecture parameters encoded in the framework demonstration

Category	Parameters
Domains	EXECUTION_MODEL, TP_CONTAINMENT, TT_SEPARATION
Channels	TP_COMM, TT_COMM, SHM_SCOPE, SPACECRAFT_IF_ACCESS
Policies	TP_COMM_CONTROL, TT_COMM_CONTROL, TP_DATA_ACCESS, TT_DATA_ACCESS, SPACECRAFT_IF_CONTROL, SHM_ACCESS, SHM_CONTROL, PLATFORM_UPDATE, TENANT_DEPLOYMENT, CODE_SIGNING, BOOT_INTEGRITY, CRYPTO_KEY_PROTECTION, RESOURCE_ISOLATION

Table 3.2: Representative allowed values from architecture_schema.yaml

Parameter	Representative allowed values	Role in the demonstration
TP_CONTAINMENT	none, logical_only, process_sandboxed, containerized, vm_isolated, partition_isolated	Describes the tenant-platform isolation boundary, affecting fragments related to tenant escape, excessive permissions, or privilege confusion.
TP_COMM_CONTROL	none, unmediated, restricted, capability_based	Describes how the tenant-platform communication is controlled, affecting fragments where a tenant reaches platform service through messages, API calls, or service requests.
SHM_SCOPE	none, tp_only, tt_only, tp_and_tt	Determines where shared memory exists in the architecture. This parameter is different from message-based communication because with shared memory, software components may influence each other through memory regions, buffers, mapped data, or shared state.
RESOURCE_ISOLATION	none, soft, hard, partitioned	Describes how strongly shared resources are limited, separated, or controlled between tenants and platform software. Examples of resources: CPU, memory, storage, I/O bandwidth, message queues, bus access, service capacity, communication buffers. This parameter affects resource-interference paths.

The use of allowed values is important because the generator cannot evaluate free-text architecture descriptions consistently. Each parameter has a predefined value set, as illustrated in Table 3.2. These values are used by the activation rules to decide whether a fragment should be included in the generated AFT and, where applicable, by the scoring method to adjust the reachability of an activated path. The full set of allowed values is kept in `architecture_schema.yaml`;

only representative examples are shown here to avoid duplicating the full schema definition in the main text.

3.2.2 `architecture_variant.yaml`

This file represents one concrete architecture configuration evaluated by the framework. While `architecture_schema.yaml` defines the available parameters and their allowed values, `architecture_variant.yaml` is the instance filled in by the analyst for a specific architecture variant, which serves as the configuration input through which the OBSW architecture to be analysed is described.

Each architecture variant assigns one value to each of the 20 parameters defined in the schema. The parameter structure is kept the same for every variant, while the selected values change according to the architecture under analysis. This is what allows two variants to be compared consistently: both are described through the same domains, channels, and policy/control parameters, but they may differ in containment, communication, shared memory, lifecycle control, resource isolation, or other relevant choices.

The file is organised into the same main categories used in the schema. The `configuration` section identifies the analysed configuration through `CONFIG_ID`. The `domains` section assigns values to execution and separation parameters such as `EXECUTION_MODEL`, `TP_CONTAINMENT`, and `TT_SEPARATION`. The `channels` section describes internal and external interaction paths, including tenant-platform communication, tenant-tenant communication, shared memory, and spacecraft-facing interface access. The `policies` section assigns values to the control mechanisms associated with those paths, such as communication control, data access, shared-memory control, update and deployment control, code signing, boot integrity, key protection, and resource isolation.

The template version of `architecture_variant.yaml` includes comments above each field showing the allowed values and, where needed, consistency constraints. These comments guide the analyst when completing the file. For example, if `TP_COMM = none`, then `TP_COMM_CONTROL` and `TP_DATA_ACCESS` should also be `none`. If tenant-platform communication is present, then its control and data-access parameters must also be assigned meaningful non-`none` values. Similar consistency constraints are included for tenant-tenant communication, shared memory, and spacecraft-facing interface access.

Before AFT generation, the completed architecture variant file is passed to the architecture validator script. The validator checks the file against `architecture_schema.yaml` by verifying that required parameters are present, that assigned values belong to the allowed value sets, and that the main consistency constraints are respected. Only after this validation step is the file treated as a valid architecture variant and used later by `aft_rules.yaml`. The activation rules then evaluate the validated parameter values to determine which fragments from `aft_fragments.yaml` are structurally applicable.

For the demonstration, two architecture variant files are instantiated from the same template. Variant A represents a stronger tenant-platform boundary, while Variant B represents a weaker con-

figuration with broader or less controlled paths between tenant software, platform services, shared resources, and lifecycle mechanisms. Because both files use the same schema, any difference in the generated AFTs can be traced back to differences in the assigned parameter values.

The actual generated and evaluated variants used in the demonstration are detailed in Section 3.3. This subsection introduces `architecture_variant.yaml` only as the configuration file completed by the analyst, validated by the architecture validator script, and then passed to the activation-rule evaluation stage.

3.2.3 `architecture_validator.py`

The architecture validator script is only an implementation support artefact used before AFT generation and its purpose is to check whether the completed `architecture_variant.yaml` file conforms to `architecture_schema.yaml`. The validator verifies that all required parameters are present, that each assigned value belongs to the allowed value set defined in the schema, and that the main consistency constraints between related parameters are respected.

This validation step is necessary because the activation rules assume a well-formed architecture variant as input. If a parameter is missing, misspelled, or assigned a value outside the schema, the generator could activate incorrect fragments or fail to evaluate the rule set. The validator therefore acts as a safeguard between the user-filled architecture description and the rule-evaluation stage.

3.2.4 `aft_fragments.yaml`

This file contains all reusable fragments defined with respect to the selected top event and `architecture_schema.yaml`. In other words, it contains the candidate BFMs, FEs, AEs, FCs and CWEs that may appear in a generated AFT for this top event, while taking into account all parameters and their values defined in the schema file, before architecture-specific activation is applied.

A fragment is a partial AFT structure, or a sub-tree, each following the metamodel by linking a BFM to one or more FEs, AEs, FCs, and CWEs. The purpose of the file is to store these reusable structures independently of any single architecture variant. Whether a fragment appears in a generated AFT is decided later by the activation rules in `aft_rules.yaml`.

In this demonstration, the fragment library covers the main architectural paths through which the tenant-platform boundary may fail. These include containment failure, communication mediation failure, authority binding failure, lifecycle integrity failure, trust-material exposure, shared-memory interference, resource interference, and indirect propagation through platform-controlled services. These categories correspond to different ways in which tenant software can move beyond its intended scope or affect trusted platform behaviour.

Table 3.3: Number of FEs and CWEs defined per BFM in the prototype implementation

Boundary Failure Mode	FEs	Located CWE instances
BFM_TP_CONTAINMENT_BOUNDARY_FAILURE	7	30
BFM_TP_ISOLATION_ACCESS_POLICY_EXPOSURE	10	53
BFM_TP_COMMAND_SERVICE_EXPOSURE	10	47
BFM_TP_TRUSTED_PLATFORM_STATE_EXPOSURE	8	44
BFM_TP_SHARED_MEMORY_EXPOSURE	6	35
BFM_TP_SOFTWARE_LIFECYCLE_EXPOSURE	10	59
BFM_TP_TRUST_MATERIAL_EXPOSURE	10	60
BFM_TP_RESOURCE_MEDIATION_EXPOSURE	10	59
BFM_TP_TT_PIVOT_EXPOSURE	8	53

The selected top event is decomposed in the fragment library into 9 BFMs and 79 FEs. Each FE is decomposed into one AE and one FC, resulting in 79 AE-FC pairs. The FCs are further linked to 440 located CWE instances, corresponding to 111 unique CWE identifiers. These values describe the complete reusable fragment library available to the generator, rather than the size of every generated AFT. For each architecture variant, only the fragments whose activation conditions are satisfied in `aft_rules.yaml` are included in the final tree.

For example, a communication mediation fragment represents a path in which tenant software interacts with a platform service through an insufficiently controlled communication interface. In this case, the attack event may be a tenant request outside its authorised scope, while the enabling fault condition may be weak authorisation, insufficient input validation, or missing service-level mediation. A resource-interference fragment follows a different logic: tenant software does not need direct command authority, but may still affect platform functions by exhausting CPU time, memory, storage, bus bandwidth, or service capacity when resource isolation is absent or weak.

Listing 3.1: Example of an FE within `BFM_TP_ISOLATION_ACCESS_POLICY_EXPOSURE` encoded in `aft_fragments.yaml`

```
FE_TP_CONTAINER_ISOLATION_POLICY_WEAKENED:
  type: FE
  label: "Container isolation policy is weakened from a tenant context"
  context: >
    The platform permits tenant software to weaken container isolation
    rules such as namespaces, capabilities, mounts, filesystem access,
    network access, or device access.
  gate: AND
  attack_event:
    id: AE_TP_ABUSES_CONTAINER_RUNTIME_POLICY_INTERFACE
    type: AE
    label: "Tenant uses a container runtime or configuration interface to change container isolation rules."
  fault_condition:
    id: FC_TP_CONTAINER_ISOLATION_POLICY_TOO_PERMISSIVE
    type: FC
    label: "Container isolation policy grants tenant software more control or access than intended."
  gate_to_cwe: OR
  cwe:
    - id: CWE-732
```

```

name: "Incorrect Permission Assignment for Critical Resource"
location: "container:runtime_configuration"
context: "Container runtime permissions, mounted paths, device nodes, or capabilities are assigned too broadly."
- id: CWE-269
  name: "Improper Privilege Management"
  location: "container:privilege_model"
  context: "Container privileges are not correctly restricted or separated from platform authority."

```

To show how the decomposition rules from Chapter 2 were applied in the prototype, Listing 3.1 presents one worked fragment from `BFM_TP_ISOLATION_ACCESS_POLICY_EXPOSURE`, which further follows the complete decomposition chain from FE, AE, FC, and localised CWE entries. By applying the BFM derivation rule, if the positive required protection is *Platform isolation and access-control policies must remain under trusted platform control and must not be modifiable, weakenable, or influenceable from tenant software*, then if we negate it, it becomes *Platform isolation or access-control policies become modifiable, weakenable, bypassable, or influenceable from tenant software*, which is exactly the meaning of the parent BFM.

The derivation of the FE under this BFM was done according to the FE derivation rule, which states that an FE must refine one BFM into a specific architecture-level path while still remaining at a higher abstraction level than AE-FC pairs. The FE in Listing 3.1, suggests a tenant-platform isolation through containerization. If the architecture relies on containers to enforce separation between tenant software and platform authority, then the container isolation policy becomes one concrete policy mechanism whose weakening can realise the broader BFM.

For this reason, the FE is defined as `FE_TP_CONTAINER_ISOLATION_POLICY_WEAKENED`. This FE represents the specific outcome in which tenant software is able to weaken container isolation rules, such as namespaces, capabilities, filesystem access, network access, or device access. It is narrower than the parent BFM because it refers specifically to container isolation policy, but it is still at architecture level because it describes what the architecture permits, rather than how a particular exploit is implemented.

After the FE has been identified, it is decomposed into an AE and FC, following the layer 3 rule from Chapter 2. The FE `FE_TP_CONTAINER_ISOLATION_POLICY_WEAKENED` can occur only when two elements are present at the same time: a tenant-originating action that attempts to exercise the path, and an architectural weakness that allows this action to become effective. This is why the FE is connected to the AE and FC through an AND gate.

The AE is defined as `AE_TP_ABUSES_CONTAINER_RUNTIME_POLICY_INTERFACE` to illustrate the untrusted software behaviour: tenant software uses a container runtime or configuration interface to change, weaken, or influence container isolation rules.

The FC is defined as `FC_TP_CONTAINER_ISOLATION_POLICY_TOO_PERMISSIVE` and it captures the enabling architectural condition: the container isolation policy grants tenant software more control or access than intended. Without this fault condition, the tenant action represented by the AE should be rejected, contained, or prevented from changing the isolation policy. Therefore, the AE alone is not sufficient to realise the FE; the failure occurs only when the tenant-originating action is combined with a weakened or overly permissive architectural control.

The final step is the selection of the localised CWE entries under the FC. Following the rule of the fourth layer decomposition, CWEs are attached as localised weakness labels, rather than as generic ones. This means that each CWE must support the parent FC and must be tied to a specific architectural location.

For the `FC_TP_CONTAINER_ISOLATION_POLICY_TOO_PERMISSIVE`, the selected CWEs explain why a container isolation policy may grant tenant software more control or access than intended. CWE-732, *Incorrect Permission Assignment for Critical Resource*, is located at `container:runtime_configuration` and represents overly broad runtime permissions, mounted paths, device nodes, or capabilities. CWE-269, *Improper Privilege Management*, is located at `container:privilege_model` and represents incorrect separation between tenant container privileges and platform authority.

The CWE entries are connected to the FC through an OR gate. This means that either weakness category can instantiate the fault condition; multiple CWEs can be present at once, but one is sufficient for the FC to be present.

3.2.5 `aft_rules.yaml`

The file `aft_rules.yaml` encodes the activation rules used by the generator. While `aft_fragments.yaml` stores the reusable AFT structures, `aft_rules.yaml` determines which of those structures are relevant for a specific architecture variant. It therefore acts as the link between the parameterised architecture input and the generated AFT.

The rule file is directly related to both the architecture schema and the fragment library. It is related to `architecture_schema.yaml` because every rule condition is written using the parameter names and allowed values defined in that schema. It is also related to `aft_fragments.yaml` because each rule refers to specific fragment identifiers that can be activated in the generated AFT. For this reason, the rule set is specific to the selected top event. A different top event would require a different fragment library, and therefore a different set of activation rules, even if some of the same architecture parameters were reused.

The encoded rule file instantiates the activation semantics in a tree-oriented form. Each BFM rule contains a set of FE rules, and each FE rule contains one nested `fc_rule`. This mirrors the structure used in `aft_fragments.yaml`, where each FE branch contains one associated AE and one associated FC. During generation, the BFM rule is evaluated first, followed by the FE rule and then the nested FC rule. The FE branch is retained only when both the FE activation condition and the associated FC activation condition are satisfied. The associated AE is then included structurally with the retained FE, because it is predefined in the fragment library as the attack-side action linked to that failure event.

`aft_rules.yaml` uses parameter aliases to keep the encoded rules readable. The architecture variant files are organised as nested YAML structures, with parameters grouped under domains, channels, and policies. Instead of writing the full nested path each time, the rule file maps short parameter names to their full locations. For example, `TP_CONTAINMENT` maps to

domains.TP_CONTAINMENT, SHM_SCOPE maps to channels.internal.SHM_SCOPE, and RESOURCE_ISOLATION maps to policies.RESOURCE_ISOLATION. The rules can then use the shorter names while the implementation still evaluates the correct values from the validated architecture variant.

For this demonstration, the activation rules are organised by BFM family. Within each, the file defines the FE rules and their associated nested FC rules. This organisation makes the rule file easier to inspect because each group corresponds to one type of attack-fault path already defined in the fragment library.

For clarity, Listing 3.2 shows one representative rule excerpt from `aft_rules.yaml`. The example activates the shared-memory exposure BFM when the architecture variant contains tenant-platform shared memory. It then activates a more specific FE when the tenant has write-capable access to that shared memory.

Listing 3.2: Excerpt of a shared-memory activation rule in `aft_rules.yaml`

```
BFM_TP_SHARED_MEMORY_EXPOSURE:
  fragment_ref: BFM_TP_SHARED_MEMORY_EXPOSURE
  activate_when:
    parameter: SHM_SCOPE
    in: [tp_only, tp_and_tt]
  rationale: >
    Tenant-platform shared-memory exposure is relevant when shared memory
    is mapped between tenant and platform domains.

fe_rules:
  FE_TP_SHARED_MEMORY_CORRUPTS_PLATFORM_STATE:
    fragment_ref: FE_TP_SHARED_MEMORY_CORRUPTS_PLATFORM_STATE
    activate_when:
      all_of:
        - parameter: SHM_SCOPE
          in: [tp_only, tp_and_tt]
        - parameter: SHM_ACCESS
          in: [write_only, read_write]
    rationale: >
      Tenant has write-capable access to T-P shared memory that can
      influence platform state.
  fc_rule:
    fragment_ref: FC_TP_SHARED_MEMORY_EXPOSES_PLATFORM_STATE
    activate_when:
      all_of:
        - parameter: SHM_SCOPE
          in: [tp_only, tp_and_tt]
        - parameter: SHM_ACCESS
          in: [write_only, read_write]
```

Listing 3.2 illustrates how an architecture condition is translated into a structural decision in the generated AFT. The BFM is activated when `SHM_SCOPE` indicates that shared memory exists between tenant and platform domains. The FE is more specific: it is activated only when the same shared-memory path exists and the tenant side has write-capable access, represented by `SHM_ACCESS = write_only` or `SHM_ACCESS = read_write`. The associated FC uses the same condition because, in this prototype, the schema does not define a finer-grained parameter for the internal weakness of the shared-memory mapping. The rule therefore treats the condition as

structurally applicable and leaves detailed implementation assessment to later analysis.

The encoded rules translate architecture values into structural decisions in the generated tree. A satisfied rule includes the referenced fragment or event, while an unsatisfied rule leaves it out. This is what allows the same fragment library to produce different AFTs for different architecture variants. The generator does not manually add or remove paths after the tree is built; the visible differences between variants come from the parameter values evaluated by `aft_rules.yaml`.

`aft_rules.yaml` is one of the main traceability artefacts of the demonstration. When a fragment appears in the AFT generated for one variant but not in the other, the difference can be traced back to a specific activation rule and to the architecture parameter values that satisfied it. This supports the purpose of the demonstration: showing that changes in the architecture description are reflected in the generated AFT structure.

3.2.6 `aft_generator.py`

The script `aft_generator.py` is the implementation component that connects the encoded artefacts and produces the generated AFT output. It is not a modelling artefact in itself, since it does not define architecture parameters, fragments, or activation conditions, but it operationalises the generation process by reading the validated architecture variant, the fragment library, and the activation rules, then assembling the architecture-specific AFT.

The script takes as input one validated architecture variant from `validated_architecture_variants` directory (i.e., `variant_1.yaml` or `variant_2.yaml`), the reusable fragment library from `aft_fragments.yaml`, and the activation rules from `aft_rules.yaml`. The top event can be selected interactively or provided through a command-line argument.

Before rule evaluation, the script flattens the nested architecture variant into a single parameter dictionary. This allows the activation conditions in `aft_rules.yaml` to be evaluated using the short parameter names defined in the rule file. The script then checks whether the selected top event exists in both the fragment library and the rule file. If the selected top event is not present in both artefacts, generation stops with an error.

The generation process follows the hierarchy of the AFT. First, the selected top event is copied from `aft_fragments.yaml`, but its BFM subtrees are initially removed. The script then evaluates the BFM rule groups defined in `aft_rules.yaml`. A BFM rule checks whether the broad architectural mechanism or attack surface represented by it is relevant to the analysed variant. If the BFM condition is satisfied, its corresponding metadata is copied from the fragment library and the script proceeds to evaluate the FE rules under that BFM.

Each FE rule is evaluated against the architecture parameter context. An FE rule checks whether a specific failure outcome is structurally applicable within the active BFM. If the FE activation condition is satisfied, the script then evaluates the nested `fc_rule` associated with that FE. The FE branch is retained only when both the FE activation condition and the associated FC activation condition are satisfied. Only after both checks are satisfied does the script copy the corresponding predefined FE branch from `aft_fragments.yaml` into the generated AFT.

The output of the script is a generated YAML file stored in `generated_afts` directory. The filename includes the variant identifier and the selected top event, making the output traceable to the analysed architecture configuration. The generated file also contains metadata recording the selected top event, the architecture variant file, the fragment file, the rule file, and the schema versions. This supports traceability from the generated AFT back to the encoded framework artefacts.

After generation, the script validates the metamodel of the generated AFT and reports warnings if expected elements are missing. It then prints a short generation summary, including the number of generated BFMs, FEs, AEs, FCs, and CWE nodes. This summary is useful for checking that generation succeeded and for comparing the structural size of AFTs generated for different architecture variants.

In the demonstration workflow, `aft_generator.py` is executed after the architecture variant has been validated because its role is to transform a schema-valid architecture configuration into an AFT instance by applying the encoded activation rules to the reusable fragment library.

3.2.7 `aft_visualizer.py`

The `aft_visualizer.py` is an implementation support file used to convert generated AFT YAML outputs into graphical tree representations. It is executed after the generation of the AFT and does not influence fragment activation, rule evaluation, or scoring. Its only purpose is to make the generated AFT structure easier to inspect and compare during the demonstration.

The script takes as input a generated AFT file from the `data/generated_afts` directory. The expected input follows the naming pattern `generated_aft_variant_<CONFIG_ID>_<TOP_EVENT>.yaml`. The visual outputs are written to the `data/visualized_afts` directory and may be exported as `png`, `svg`, or `pdf` files. The script can be used interactively, by selecting from the available generated AFT files, or non-interactively by providing the variant identifier, top event, and optional BFM identifier through command-line arguments.

The visualisation can be generated in two modes. The first mode renders the full AFT for the selected top event, giving a complete view of all activated BFMs and FE branches for one architecture variant. The second mode renders a single BFM subtree. This is useful when the full generated AFT is too large to inspect clearly, or when a specific attack-fault path family needs to be discussed in more detail.

The script uses a consistent graphical notation for the AFT metamodel. The top event, BFMs, FEs, and FCs are represented as rectangles. AEs are represented as diamonds, and CWEs are represented as circular basic-event nodes. AND and OR gates are drawn directly as logic-gate symbols using Matplotlib. The tree uses orthogonal connectors to show the parent-child relationships between the generated nodes.

To keep the figures readable, the visual tree displays node identifiers rather than full descriptions. CWE nodes display only the CWE ID. Long identifiers are wrapped inside their shapes, while detailed descriptions, meanings, rationales, CWE names, locations, and contexts remain available in the generated YAML output. The YAML file therefore remains the traceable source of the gen-

erated AFT structure, while the visualisation provides a readable representation for interpretation and discussion.

The graphical trees produced for the evaluated architecture variants are shown in Section 3.3, where each variant is presented and discussed. The visualisations therefore complement the generated YAML files and support the comparison of the activated AFT structures across architecture variants.

3.2.8 `aft_likelihood_analyzer.py`

The `aft_likelihood_analyzer.py` is an implementation support file used for the likelihood evaluation phase of the framework. It is executed after an architecture-specific AFT has been generated and provides the link between the generated tree structure and the likelihood scoring method defined in Section 2.4. The script does not change the generated AFT structure, it only extracts the activated FEs from the generated AFT and supports the assignment and propagation of likelihood scores.

The script has two main functions. First, it creates an Excel likelihood-scoring template from a generated AFT YAML file. The template contains one row for each activated FE branch and includes the corresponding variant ID, top event ID, BFM ID, FE ID, AE ID, and FC ID. The analyst then fills in only two values for each row: the AE likelihood and the FC likelihood. These values are assigned on the ordinal 1-5 likelihood scale used by the framework.

Second, after the scoring template has been completed, the script reads the filled Excel file and computes the likelihood values for the generated AFT. For each FE, the AE likelihood and FC likelihood are multiplied to obtain a raw FE likelihood product. This raw value is then mapped back onto the 1-5 ordinal scale. The resulting FE likelihoods are aggregated upward through the tree. For each BFM, the likelihood is calculated as the maximum likelihood of its retained FE branches. The top event likelihood is then calculated as the maximum likelihood of the activated BFMs.

The implemented propagation method is therefore consistent with the semi-quantitative scoring approach of the framework, where the FE level captures the combined contribution of the attack action and the enabling fault condition, while the BFM and TE levels use the maximum operation to represent alternative sufficient paths. The output is an ordinal likelihood score, not an absolute probability of top event occurrence.

The script also preserves traceability information during the analysis. CWE nodes are included in the generated report for classification and traceability, but they do not receive likelihood scores and do not affect likelihood propagation. This reflects their role in the framework: CWEs describe the type of weakness associated with an activated FC, while the AE and FC scores determine the FE likelihood.

The outputs of the script are written to the `data/analysis_reports` directory. The Excel report provides a readable summary of the analysis, including the top event likelihood, BFM likelihoods, FE likelihoods, dominant BFMs, dominant FEs, missing or invalid scores, and CWE traceability. A YAML report is also generated to preserve the computed results in a machine-readable

format. These outputs support the comparison of architecture variants by showing both the final likelihood score and the activated branches that dominate the result.

The generated AFT provides the activated structure, the likelihood analyzer provides the semi-quantitative evaluation of that structure, and the resulting reports are used to compare the evaluated architecture variants.

3.2.9 `aft_comparator.py`

The `aft_comparator.py` is an implementation support file used to compare two generated architecture-specific AFTs. Its purpose is to make the structural differences between architecture variants explicit, especially when the complete generated trees are too large to compare visually in the main text.

In the demonstration, the comparator is used to compare the generated AFT for Variant 1 with the generated AFT for Variant 2 under the same selected top event. The script reads the two generated AFT YAML files from the `generated_afts` directory and produces a comparison report, showing which architecture parameters, BFM, FEs, and CWE basic-event instances are common to both variants and which appear only in one variant.

The comparison is performed at four levels. The first level compares the architecture parameters stored in the generated AFT files. This identifies which parameter values are the same and which differ between the two evaluated variants. The second level compares the activated BFMs by BFM identifier. The third level compares the retained FEs under each BFM, using the pair formed by the BFM identifier and FE identifier. The fourth level compares the CWE instances associated with the retained FCs. In this last comparison, a CWE instance is identified by the BFM, FE, CWE identifier, and architectural location. This means that the same CWE classification appearing at different architectural locations is treated as a distinct basic-event instance.

The main output of the comparator is an Excel report stored in the `data/comparison_reports` directory. The report contains separate sheets for the summary, architecture parameters, BFM comparison, FE comparison, and CWE basic events. The comparison status indicates whether each item is common to both variants, appears only in Variant A, appears only in Variant B, or has different values. The script can also write an optional YAML comparison report, which preserves the same comparison results in a machine-readable form.

The comparator supports an iterative architecture-refinement process, where, for example, Variant A represents an initial architecture configuration, while Variant B represents a modified configuration in which one or more architecture parameter values have been changed. These changes may correspond to stronger containment, stricter communication mediation, stronger lifecycle controls, improved key protection, or harder resource isolation. For example, an architecture refinement may change `CODE_SIGNING` from `optional` to `enforced`, `RESOURCE_ISOLATION` from `soft` to `hard`, or `TP_CONTAINMENT` from `containerized` to `vm_isolated`.

The comparator does not automatically select or recommend mitigations. Its role is to show the structural effect of architecture changes on the generated AFTs. A design change is represented

by modifying the values in the architecture variant file and regenerating the AFT. The comparator then identifies which BFM, FEs, and CWE basic-event instances are common to both variants, which are removed, and which are added. If likelihood analysis has also been performed, the comparison can be interpreted together with the likelihood reports to understand how the architecture change affects the dominant attack-fault paths.

In this sense, the comparator supports trade-off analysis and architecture refinement. It helps the analyst see whether a proposed architecture change removes relevant attack-fault paths, introduces new ones, or shifts the dominant exposure to another part of the architecture. The result can then inform further refinement of the architecture parameters.

Figure 3.1 illustrates this use of the comparator. Two architecture variants are generated and analysed under the same selected top event. The comparator then compares the resulting AFTs and highlights parameter differences, added or removed BFMs and FEs, CWE differences, and, where likelihood analysis is available, changes in the analysed likelihood results. The figure presents the comparator as part of an iterative refinement loop, where the comparison results may guide later updates to the architecture variant file.

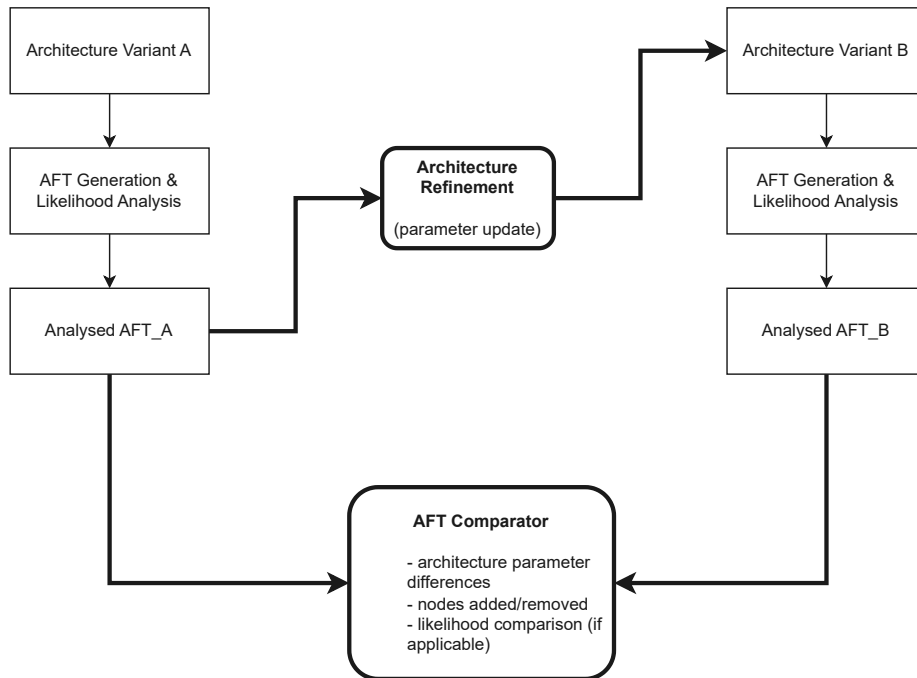


Figure 3.1: *Comparator-supported architecture refinement process*

3.3 Evaluated Architecture Variants

The purpose of this section is to define the two architecture configurations used to demonstrate the framework. Differences in the generated AFTs come from changes in the architecture

parameters, while the analysed top event is kept constant.

The two variants were chosen to represent two different points in the design space. Variant 1 represents a stronger tenant-platform boundary configuration, with a microkernel execution model, VM-based tenant-platform containment, platform-mediated communication, no shared memory, stronger lifecycle controls, hardware-rooted boot integrity, hardware-backed key protection, and hard resource isolation. Variant 2 represents a weaker and more exposed configuration, with a monolithic kernel OS, container-based tenant-platform containment, direct APIs, tenant-to-tenant shared memory, softer resource isolation, and weaker lifecycle and integrity controls.

These two configurations are used as contrasting examples that activate different sets of AFT fragments. They allow the demonstration to show how architectural choices are translated into different attack-fault tree structures and different semi-quantitative likelihood results.

3.3.1 Variant 1: Microkernel-Based VM/Container SataaS Architecture with Platform-Mediated Tenant Communication

This architecture configuration describes a multi-tenant OBSW architecture in which untrusted tenant applications run on the same OBC as trusted platform software, while being isolated from it through a VM boundary. The underlying execution model is a microkernel, meaning that only minimal privileged services run in kernel space, while higher-level platform services remain outside the kernel in order to reduce the trusted computing base (TCB).

In this variant, tenant applications are deployed inside a tenant VM environment, and each tenant application runs inside its own container. The main tenant-platform boundary is therefore VM-based, while tenant-to-tenant separation is container-based, meaning that tenants are not merged into one shared execution environment.

Tenant applications do not communicate directly with trusted platform internals. This communication is done through a platform-mediated API provided by trusted platform software. The communication control is restricted, meaning that tenant requests are subject to platform-defined restrictions such as access control, request filtering, operation allowlists, validation, or rate control. Tenants have command-level access through the platform-mediated API, meaning that they may request platform-mediated operations such as payload tasking, data processing, or data downlink. However, they cannot directly command spacecraft-facing interfaces.

Tenant-to-tenant communication is also not direct. When one tenant needs to exchange data with another tenant, the communication is routed through a platform-brokered API. This allows read/write data exchange between tenants, but only under platform mediation. There is no direct tenant-to-tenant channel and no shared memory between tenants or between tenants and the platform.

The configuration also assumes a controlled SataaS lifecycle. Tenant software can be changed after deployment, but only through a platform-controlled deployment process. Tenant applications must pass validation, scheduling, and activation checks before being deployed or replaced. Code signing is enforced, boot integrity is hardware-rooted, and cryptographic keys are stored in

hardware-backed secure storage.

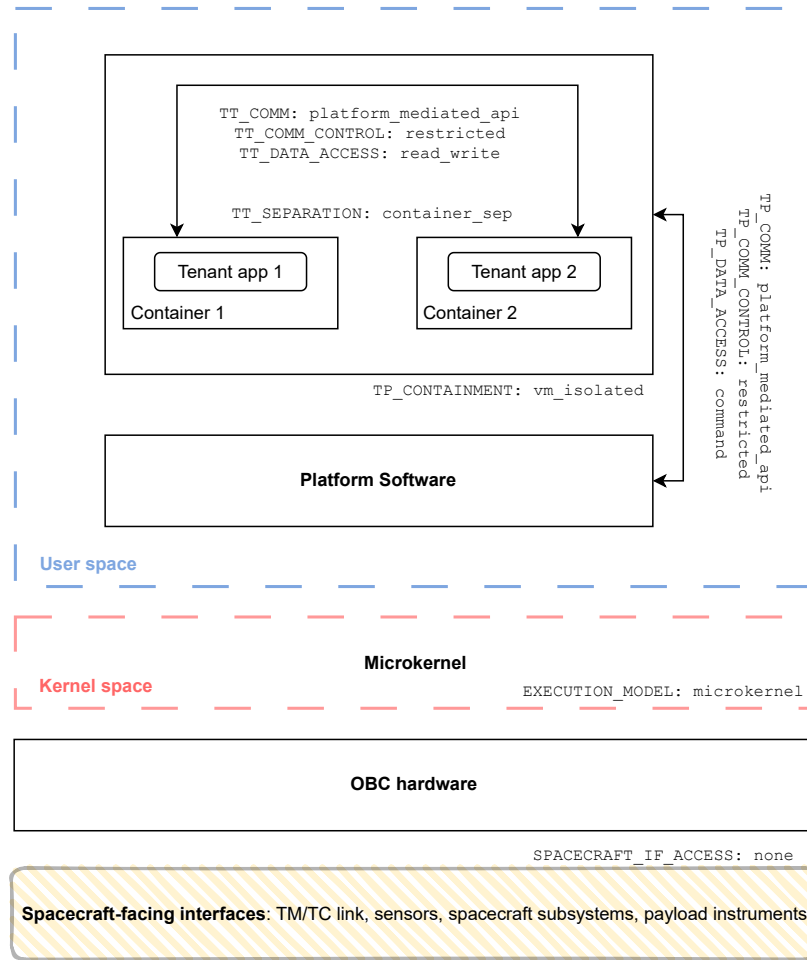


Figure 3.2: *Variant 1: Microkernel-Based VM/Container SataaS Architecture with Platform-Mediated Tenant Communication*

3.3.2 Variant 2: Monolithic Containerized SataaS Architecture with Direct Tenant APIs and Tenant Shared Memory

Variant 2 describes a multi-tenant OBSW architecture in which untrusted tenant applications run on the same OBC as trusted platform software under a monolithic execution model. In contrast to Variant 1, this architecture does not introduce a VM boundary between the tenant domain and the trusted platform. Tenant applications are contained directly through container-based isolation mechanisms, and each tenant application runs in its own container. Therefore, container boundaries provide both the tenant-platform containment mechanism and the tenant-tenant separation mechanism.

Tenant-platform communication occurs through a direct API. The communication control is

restricted, meaning that tenant requests are subject to platform-defined checks such as access control, validation, request filtering, or operation restrictions. Tenants here also have command-level access through this API, meaning that they may request platform operations through the tenant-platform interface. In addition, the architecture includes direct read-only access to spacecraft-facing interfaces. This does not give tenants direct command authority over spacecraft-facing functions, but it does introduce a direct observation path to spacecraft-facing data.

Tenant-to-tenant communication is not platform-brokered in this configuration. Tenants exchange data through direct tenant-facing APIs with restricted read/write access. In addition, a tenant-to-tenant shared-memory region is present and it is configured with read/write access and no runtime mediation.

In a SataaS mission, tenant-to-tenant shared memory may represent a direct mapped-memory buffer used for high-throughput onboard data exchange between tenant applications. For example, one tenant may perform payload data preprocessing, while another tenant performs compression, classification, or AI-based analysis on the intermediate output. This design reduces copying and latency compared with API-mediated exchange, but it also introduces direct cross-tenant read/write paths. If the shared memory region is unmediated, a tenant may corrupt, overwrite, or read data belonging to another tenant, creating attack-fault paths related to shared-memory access-control failure, synchronization errors, and cross-tenant interference.

Compared to Variant 1, the lifecycle and protection mechanisms are weaker. Platform updates and tenant deployment are authenticated, but not fully controlled. Code signing is optional, boot integrity is software-based, cryptographic keys are protected through software mechanisms, and resource isolation is soft rather than hard.

A practical realization of this type of architecture could be based on a general-purpose monolithic OS, such as Linux, combined with container technologies such as Docker.

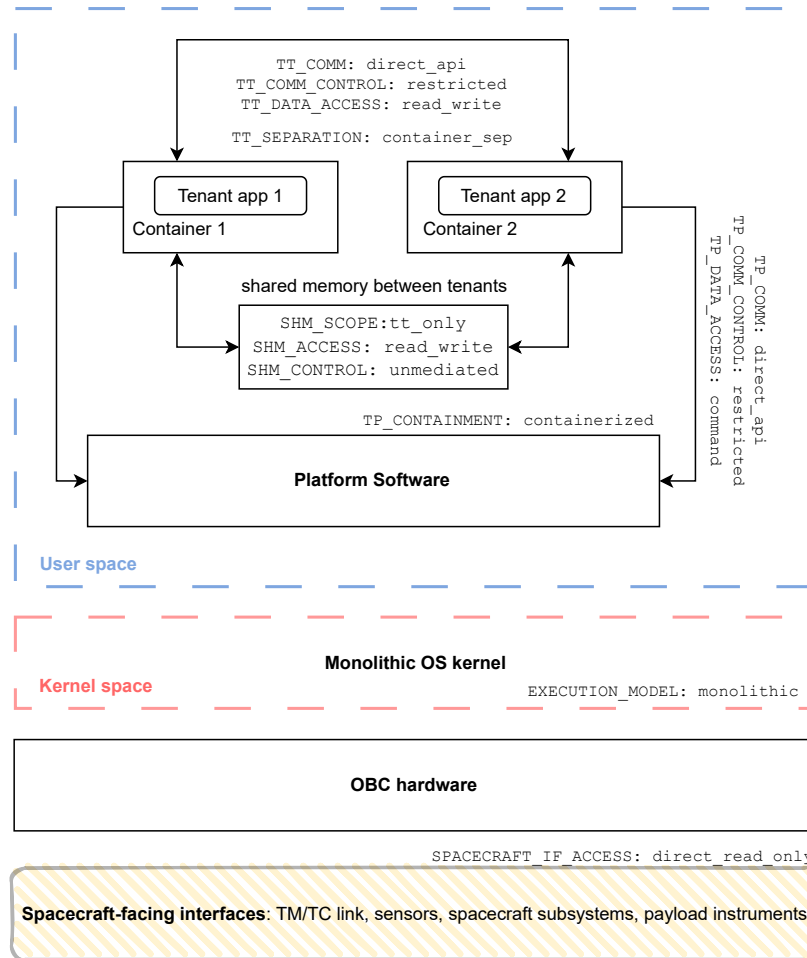


Figure 3.3: *Variant 2: Monolithic Containerized SataaS Architecture with Direct Tenant APIs and Tenant Shared Memory*

3.4 Generated AFTs

The complete generated trees are too large to include as full figures in the main text while preserving readability. For this reason, this section presents the generated results through summary tables and interpretation of the retained fragment families. Selected BFM-level visualisations can be included where they are useful for illustrating specific paths. The complete YAML files remain the traceable source of the generated AFT structures.

Table 3.4: Generated AFT size for the evaluated architecture variants

Variant	BFMs	FEs	AEs	FCs	CWEs	Main observation
Variant 1	7	41	41	41	227	The generated AFT remains focused on containment, command-service, platform-state, trust-material, resource-mediation, and tenant-to-tenant pivot paths. Stronger lifecycle controls and the absence of tenant-platform shared memory reduce the retained structure.
Variant 2	8	62	62	62	348	The generated AFT is larger because the monolithic execution model, container-based containment, weaker lifecycle controls, software-based key protection, soft resource isolation, and tenant-to-tenant shared memory activate additional branches.

The difference between the two generated AFTs shows how the same top event can lead to different structural outputs when the architecture parameters change. Variant 1 activates fewer FE branches because several controls constrain the relevant attack-fault paths. Variant 2 activates more FE branches because additional architecture-level exposure conditions are present.

3.5 Analysis and Comparison of the Generated AFTs

The comparison is performed in two steps. First, the generated AFTs are compared structurally by examining the retained BFMs and FEs, showing which attack-fault paths are applicable to each architecture variant according to the activation rules. Second, the likelihood-analysis outputs are compared to identify the dominant BFMs and FEs for each variant. The likelihood scores are ordinal values used for relative comparison between architecture variants, rather than calibrated probabilities.

3.5.1 Structural Comparison

Table 3.5 compares the generated AFTs at BFM level. Variant 1 contains 41 retained FE branches, while Variant 2 contains 62 retained FE branches. The difference of 21 additional FE branches in Variant 2 shows that the weaker architecture configuration activates a broader set of attack-fault paths.

Table 3.5: Structural comparison of generated AFTs by BFM family

BFM family	Variant 1 FEs	Variant 2 FEs	Difference
Containment boundary failure	1	2	+1
Isolation and access-policy exposure	6	6	0
Command-service exposure	10	10	0
Trusted platform state exposure	8	8	0
Software lifecycle exposure	0	10	+10
Trust-material exposure	4	9	+5
Resource-mediation exposure	5	10	+5
Tenant-to-tenant pivot exposure	7	7	0
Total	41	62	+21

The first difference appears in the containment boundary BFM. Variant 1 uses a microkernel execution model with VM-based tenant-platform containment. Its generated containment branch therefore focuses on VM escape reaching platform authority. Variant 2 uses a monolithic execution model with container-based tenant-platform containment. Its generated containment BFM retains both a container escape path and a shared privileged TCB compromise path.

The isolation and access-policy BFM is active in both variants with the same number of retained FE branches. However, the architectural meaning differs. In Variant 1, the relevant paths are associated with VM isolation and platform-mediated communication. In Variant 2, they are associated with container isolation and direct API access. This shows that a BFM family may remain active across variants while its specific branch content reflects the architectural mechanism used in each case.

The command-service and trusted platform state BFMs are also retained in both variants with the same number of FE branches. This is expected because both variants provide tenant-platform communication and command-level tenant access. Variant 1 provides this access through a platform-mediated API, while Variant 2 provides it through a direct API. In both cases, tenant-originated requests can reach platform command-handling or state-handling logic. Therefore, command authorization, command authentication, malformed command handling, replay, injection, confused-deputy behaviour, and trusted platform state influence remain structurally relevant in both generated AFTs.

The largest structural difference is the software lifecycle BFM. This BFM is absent from Variant 1 and active in Variant 2 with 10 retained FE branches. Variant 1 uses controlled platform updates, platform-controlled tenant deployment, enforced code signing, hardware-rooted boot integrity, and hardware-backed key protection. These parameter values prevent the software lifecycle BFM from being retained. Variant 2 uses authenticated update and deployment processes, optional code signing, software-based boot integrity, and software-based key protection. These values activate lifecycle-related paths such as malicious platform update installation, tenant software activation with platform authority, code-signature bypass, integrity-check bypass, boot-chain compromise, rollback, lifecycle-policy downgrade, injected lifecycle-control data, unauthorized code paths, and

software activation interface abuse.

The trust-material BFM is active in both variants, but it is larger in Variant 2. Variant 1 retains four FE branches, while Variant 2 retains nine. This difference is mainly caused by the weaker trust-material protection in Variant 2. Software-based key protection and optional signing make additional trust-material exposure paths structurally applicable. These include paths related to platform key access, key use, key extraction from software storage, side-channel inference, and weak or default trust material.

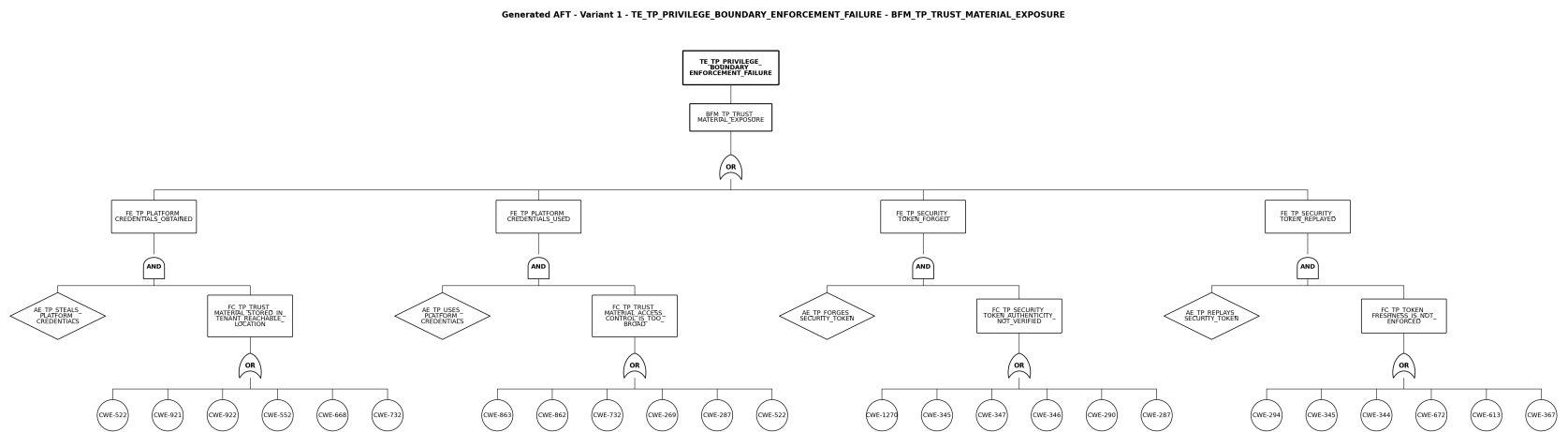


Figure 3.4: *Generated trust-material exposure BFM for Variant 1*

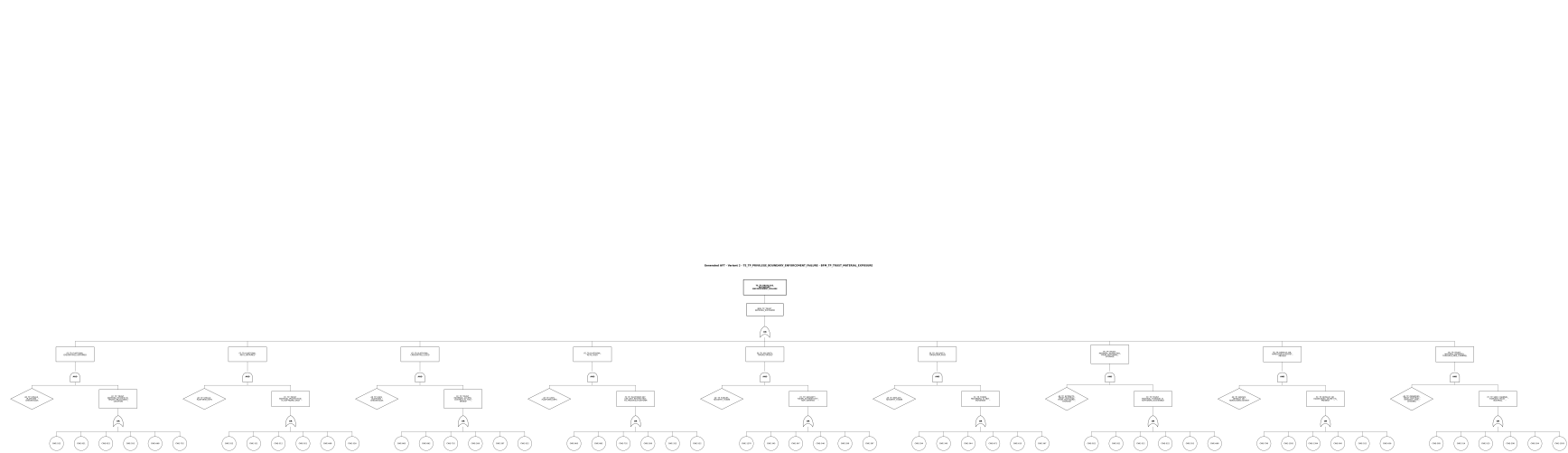


Figure 3.5: *Generated trust-material exposure BFM for Variant 2*

Figures 3.4 and 3.5 illustrate the difference in the generated trust-material exposure subtree. Variant 1 retains a smaller trust-material subtree, reflecting the stronger protection mechanisms encoded in its architecture parameters. Variant 2 retains a larger subtree because additional trust-material paths become structurally applicable when key protection and signing controls are weaker.

The resource-mediation BFM also increases from five retained FEs in Variant 1 to ten retained FEs in Variant 2. This follows from the difference between hard and soft resource isolation. Hard resource isolation in Variant 1 constrains several resource-interference paths. Soft resource isolation in Variant 2 activates additional paths involving resource-limit bypass, fail-open mediation, scheduling manipulation, resource-state influence, and mediation races.

The tenant-to-tenant pivot BFM remains active in both variants. This is relevant because both architectures include tenant-to-tenant interaction and tenant-platform authority paths. In Variant 1, the pivot paths are associated with platform-mediated tenant-to-tenant communication. In Variant 2, tenant-to-tenant shared memory provides an additional architectural basis for cross-tenant influence before platform-facing authority is reached. The retained FE count is the same, but the architectural explanation differs because the interaction mechanism changes.

3.5.2 Likelihood Comparison

The likelihood comparison adds a second layer of interpretation by identifying which retained paths dominate the generated AFTs after AE and FC scores are applied. The likelihood analysis uses the scoring method defined in the framework: each FE likelihood is calculated from the AE and FC likelihood scores, BFM likelihood is computed as the maximum FE likelihood under that BFM, and the TE likelihood is computed as the maximum BFM likelihood.

Table 3.6 summarises the top-event likelihood results for the two variants.

Table 3.6: Top-event likelihood comparison

Variant	Top-event likelihood	Dominant BFM families
Variant 1	3 - Medium	Isolation and access-policy exposure; command-service exposure; resource-mediation exposure.
Variant 2	5 - Very High	Containment boundary failure; command-service exposure; trust-material exposure; resource-mediation exposure; tenant-to-tenant pivot exposure.

The top-event likelihood provides a headline likelihood category for the selected top event, but it should not be used alone to fully rank architecture variants. Since the framework uses a 1–5 ordinal scale and propagates likelihood through OR gates using the maximum operation, the top-event score can saturate at the highest category. Once a variant reaches a top-event likelihood of 5, a more exposed variant cannot receive a higher top-event score. Therefore, variants with the same top-event likelihood must be differentiated using the underlying likelihood profile, including

the number of BFM families and FEs reaching high or very high likelihood, the dominant FE paths, and the architecture parameters responsible for those paths.

Variant 1 reaches a top-event likelihood of 3, corresponding to a Medium likelihood level. The dominant BFM families are isolation and access-policy exposure, command-service exposure, and resource-mediation exposure. This means that the highest-scoring paths in Variant 1 are not the VM containment escape path, but the paths where tenant-originated requests can influence policy, command handling, or resource mediation.

Variant 2 reaches a top-event likelihood of 5, corresponding to a Very High likelihood level. The dominant BFM families are containment boundary failure, command-service exposure, trust-material exposure, resource-mediation exposure, and tenant-to-tenant pivot exposure. This shows that the weaker architecture parameters in Variant 2 do not only increase the number of retained FE branches; they also increase the maximum likelihood values propagated to the BFM and top-event levels.

Table 3.7 compares the BFM likelihood values for both variants.

Table 3.7: BFM likelihood comparison

BFM family	V1 FEs	V2 FEs	V1 likelihood	V2 likelihood
Containment boundary failure	1	2	1 - Very Low	5 - Very High
Isolation and access-policy exposure	6	6	3 - Medium	4 - High
Command-service exposure	10	10	3 - Medium	5 - Very High
Trusted platform state exposure	8	8	2 - Low	4 - High
Software lifecycle exposure	0	10	-	4 - High
Trust-material exposure	4	9	2 - Low	5 - Very High
Resource-mediation exposure	5	10	3 - Medium	5 - Very High
Tenant-to-tenant pivot exposure	7	7	2 - Low	5 - Very High

The likelihood comparison shows that structural size alone does not explain the difference between the two variants. Some BFM families have the same number of retained FE branches in both variants, but different likelihood values. For example, command-service exposure contains 10 retained FE branches in both variants, yet its likelihood increases from Medium in Variant 1 to Very High in Variant 2. This occurs because Variant 2 uses a more exposed communication configuration, and the AE and FC scores assigned to the retained command-service paths are higher.

The same pattern appears in trusted platform state exposure and tenant-to-tenant pivot exposure. Both BFMs have the same number of retained FE branches in Variant 1 and Variant 2, but the likelihood values are higher in Variant 2. This shows that the framework comparison should consider both retained structure and scored accessibility. The generated structure identifies which paths are applicable, while the likelihood analysis identifies which applicable paths are more dominant for the evaluated architecture.

The containment boundary BFM shows a strong difference in both structure and likelihood. Variant 1 retains one containment FE and receives a Very Low likelihood score. Variant 2 retains two containment FEs and reaches a Very High likelihood score. The dominant containment FE in Variant 2 is the monolithic TCB compromise path. This reflects the effect of the monolithic execution model, where tenant-triggerable behaviour can interact with a broader privileged execution base.

The software lifecycle BFM is absent from Variant 1 and active in Variant 2 with a High likelihood score. This confirms the effect of the lifecycle-related parameters. Controlled update and deployment, enforced code signing, and hardware-rooted boot integrity prevent this BFM from being structurally retained in Variant 1. In Variant 2, authenticated but less controlled lifecycle mechanisms, optional code signing, and software-based boot integrity create structurally applicable lifecycle paths with higher likelihood scores.

The trust-material BFM increases from Low in Variant 1 to Very High in Variant 2. The dominant trust-material FE in Variant 2 is trust material extracted from software storage. This result is consistent with software-based key protection, which makes trust material more accessible than hardware-backed key storage. The resource-mediation BFM also increases from Medium to Very High because soft resource isolation allows fail-open mediation, queue flooding, and shared resource state influence to become more dominant.

Overall, Variant 2 is both structurally broader and more likely at the top-event level. Structurally, it contains 21 more retained FE branches than Variant 1. In the likelihood analysis, its top-event likelihood increases from Medium to Very High. The dominant Variant 2 paths are linked to monolithic execution, direct command-service exposure, software-accessible trust material, soft resource isolation, and tenant-to-tenant shared memory. Variant 1 remains lower because stronger containment, lifecycle, trust-material, and resource-isolation choices reduce both the number of applicable paths and the maximum propagated likelihood values.

Chapter 4

Discussion

4.1 Interpretation of the Demonstration Results

The demonstration shows that the proposed framework can generate different AFT structures from different architecture variants while keeping the same selected top event. This is an important result because it confirms that the generated tree is not fixed only by the top event. It also depends on the architecture parameter values that describe containment, communication, authority, lifecycle control, trust-material protection, shared memory, and resource isolation.

Variant 1 produced a smaller generated AFT than Variant 2. This result is consistent with its stronger architecture parameters, including VM-based tenant-platform containment, platform-mediated communication, controlled update and deployment processes, enforced code signing, hardware-rooted boot integrity, hardware-backed key protection, and hard resource isolation. These controls reduced the number of structurally applicable FE branches. The additional branches in Variant 2 were mainly caused by the monolithic execution model, container-based tenant-platform containment, direct API access, weaker lifecycle controls, optional code signing, software-based key protection, soft resource isolation, and tenant-to-tenant shared memory. These parameters made additional attack-fault paths structurally applicable, especially in the software lifecycle, trust-material, resource-mediation, and containment-related BFM families.

A larger generated tree does not automatically mean a higher likelihood for every branch. Some BFM families may have the same number of retained FEs in both variants, while still receiving different likelihood scores because the accessibility of the attack event or the weakness of the fault condition differs between architectures.

4.2 Value of Parameter-Driven AFT Generation

The main value of the framework is that it connects architecture-level design choices to generated safety-security structures. Instead of building a separate attack-fault tree manually for each architecture variant, the analyst defines a reusable fragment library and a set of activation rules.

The architecture variant then determines which fragments are retained.

This improves repeatability because the same rule set can be applied to multiple variants. It also improves traceability because each generated node can be traced back to the activation rule that retained it, and each rule can be traced back to the architecture parameters used in its condition. The framework therefore makes the reasoning behind the generated AFT explicit.

The approach is especially useful during early design, when detailed implementation evidence may not yet be available. At this stage, the analyst may still need to compare architecture options, identify exposure paths, and understand how design choices affect the safety-security model. The proposed framework supports this by using architecture parameters as the main driver of AFT generation.

4.3 Structural Comparison versus Likelihood Comparison

The framework separates structural applicability from likelihood evaluation. Structural generation answers the question: *is this attack-fault path relevant for the analysed architecture?* Likelihood analysis answers a different question: *among the structurally applicable paths, which ones appear more accessible or weakly controlled?*

This separation is useful because it avoids treating activation as proof that an attack or fault will occur. Activation only means that the fragment is meaningful for the architecture. The likelihood score is then added as a separate semi-quantitative interpretation layer.

In the demonstration, this distinction was visible in the comparison between the two variants. Some BFM families appeared in both generated AFTs, such as command-service exposure and trusted platform state exposure. This means that both architectures contain tenant-platform paths through which tenant-originated requests can reach platform logic. However, the likelihood levels differed because the two variants expose those paths through different architectural mechanisms and control strengths.

4.4 Reusability of the Framework

The framework is reusable in two ways. First, the architecture schema can be reused to describe several architecture variants using the same parameter vocabulary, allowing variants to be compared consistently. Second, the fragment library and activation rules can be reused across variants for the same selected top event.

The framework can also be extended to other top events by defining additional top-event-specific fragment sets and activation rules. The architecture schema does not need to be completely redesigned for every top event, as long as the existing parameters capture relevant architectural conditions. However, new top events may require new BFMs, FEs, FCs, and rules if they concern different types of safety-security propagation.

This means that the reusable part of the framework is the method: parameterised architecture input, reusable fragments, rule-based activation, generated AFT structure, and optional likelihood analysis. The specific demonstration is focused on the selected top event of tenant-platform privilege boundary enforcement failure.

4.5 Limitations

The limitations of this thesis can be divided into two categories. The first category concerns the proposed framework as a method for architecture-driven AFT generation. These limitations relate to the type of reasoning the framework supports and the kind of conclusions that can be drawn from its outputs. The second category concerns the current implementation used in the demonstration.

4.5.1 Framework-Level Limitations

A first framework-level limitation is that the method depends on expert-defined modelling knowledge. The framework provides a structured way to encode fragments, activation rules, and traceability links, but the initial definition of these elements still requires expert judgement. The generated AFT is therefore only as complete and accurate as the fragment library, architecture schema, and activation rules used to produce it. If a relevant failure mode or activation condition is missing, the generated tree may omit a structurally relevant path.

A second framework-level limitation is that the method supports architecture-level reasoning rather than implementation assurance. The framework can represent design choices such as containment type, communication route, lifecycle control, key protection, shared-memory scope, and resource isolation. However, it cannot verify whether a specific implementation correctly enforces those mechanisms. For example, the framework can represent that code signing is enforced, but it cannot verify the correctness, placement, or bypass resistance of the implemented signature-checking logic.

A third framework-level limitation concerns likelihood interpretation. The framework uses semi-quantitative ordinal scores to support relative comparison between architecture variants. These scores help identify dominant attack-fault paths, but, ultimately, the likelihood result depends on the scoring scale, the assigned AE and FC values, and the assumptions made during scoring.

A further limitation of the ordinal max-based likelihood aggregation is score saturation. Once the top event reaches likelihood level 5, more exposed variants cannot be distinguished by the top-event score alone. For this reason, the framework reports and interprets BFM-level and FE-level likelihood profiles in addition to the top-event score.

A fourth framework-level limitation is that the generated AFT is tied to the selected top event. The architecture schema may be reusable across analyses, but each top event requires an appropriate fragment set and activation rules. Therefore, applying the framework to a different top event

requires checking whether the existing fragments and rules are sufficient or whether new ones must be defined.

4.5.2 Implementation-Level Limitations

A first implementation-level limitation is the scope of the encoded fragment library. The prototype includes the fragments needed to demonstrate the selected top event, *Tenant-Platform Privilege Boundary Enforcement Failure*. It does not claim to contain an exhaustive library for all possible multi-tenant onboard software failures or all possible space-system safety-security scenarios.

A second implementation-level limitation is the scope of the demonstration. The implementation is evaluated using two architecture variants. This is sufficient to demonstrate the workflow, traceability, structural comparison, and likelihood comparison, but broader evaluation would require additional architecture variants, additional selected top events, and expert-reviewed case studies.

A third implementation-level limitation is that the current scripts are prototype research tools. They support validation, generation, visualisation, likelihood scoring, and comparison, but they are not developed as a complete industrial toolchain. For example, the workflow still requires manual preparation of architecture variant files, manual review of generated outputs, and manual assignment of AE and FC likelihood scores.

A fourth implementation-level limitation concerns visualisation. The generated AFTs can become large, especially for more exposed architecture variants. As a result, complete tree visualisations may be difficult to include in the main text while preserving readability. The implementation addresses this by supporting BFM-level visualisations and generated YAML outputs, but large trees still require selective presentation.

These limitations define the intended interpretation of the thesis results. The proposed framework supports early architecture-phase reasoning, traceable AFT generation, structural comparison, and semi-quantitative likelihood comparison. The current implementation demonstrates this workflow for a selected top event and two architecture variants, while leaving broader validation, tool hardening, and extension to additional top events for future work.

Conclusion

This thesis proposed and demonstrated a framework for architecture-driven generation of attack-fault trees for multi-tenant onboard software architectures. The motivation for the work was the need to support early safety-security reasoning in software-intensive spacecraft systems, where architectural choices can influence how cyber and software-fault conditions propagate toward mission-relevant consequences.

The present work first reviewed existing work on attack trees, fault trees, attack-fault trees, automated tree generation, and space-system threat modelling. This review showed that existing methods provide useful foundations for structured threat and safety-security analysis, but that a gap remains for early architecture-phase generation of AFTs for multi-tenant onboard software. In particular, existing approaches do not directly show how changes in containment, communication, lifecycle control, trust material, shared resources, and tenant-platform authority affect the generated attack-fault structure.

To address this gap, this thesis defined a framework based on four main elements: a parameterised architecture schema, reusable AFT fragments, activation rules, and semi-quantitative likelihood analysis. The architecture schema provides a controlled vocabulary for describing architecture variants. The fragment library defines reusable top-event, BFM, FE, AE, FC, and CWE structures. The activation rules determine which fragments are structurally applicable for a given architecture variant. The likelihood-analysis step then supports relative comparison between variants by scoring attack events and fault conditions and propagating the resulting values through the generated tree.

The implementation demonstrated the framework using two architecture variants. Variant 1 represented a stronger tenant-platform boundary configuration with VM-based containment, platform-mediated communication, controlled lifecycle mechanisms, enforced code signing, hardware-rooted boot integrity, hardware-backed key protection, and hard resource isolation. Variant 2 represented a more exposed configuration with a monolithic execution model, container-based containment, direct APIs, tenant-to-tenant shared memory, weaker lifecycle controls, software-based key protection, and soft resource isolation.

The generated AFT for Variant 1 contained 7 BFMs, 41 FEs, 41 AEs, 41 FCs, and 227 CWE basic-event instances. The generated AFT for Variant 2 contained 8 BFMs, 62 FEs, 62 AEs, 62 FCs, and 348 CWE basic-event instances. The structural comparison showed that Variant 2 activated additional branches mainly in software lifecycle exposure, trust-material exposure, resource-

mediation exposure, and containment boundary failure. The likelihood comparison further showed that Variant 1 reached a Medium top-event likelihood, while Variant 2 reached a Very High top-event likelihood. This confirmed that Variant 2 was both structurally broader and more dominant in the semi-quantitative likelihood analysis.

Main Contributions

The first contribution of the thesis is the definition of an architecture-driven AFT generation framework for multi-tenant OBSW. The framework links architecture parameters to attack-fault tree fragments through explicit activation rules, making the generated model traceable: each retained node can be related back to the rule that activated it and to the architecture parameters used by that rule.

The second contribution is the separation between architecture description, fragment definition, activation logic, and likelihood scoring, which makes the framework easier to inspect and extend. Architecture variants can be modified without reconstructing the whole tree, fragments can be reused across variants, and likelihood scoring can be applied after structural generation.

The third contribution is the demonstration of comparative AFT generation for two architecture variants under the same selected top event. By keeping the top event constant, the implementation showed that differences between the generated AFTs could be traced to differences in architecture parameters, supporting the use of the framework as an early design-support method for comparing architecture alternatives.

The fourth contribution is the implementation of a prototype toolchain for validation, generation, visualisation, comparison, and likelihood analysis. The encoded artefacts and scripts demonstrate that the proposed method can be operationalised in a repeatable workflow. The generated YAML files provide complete and traceable AFT structures, while the visualisations, comparator outputs, and likelihood reports support interpretation of the results.

Final Remarks and Future Work

The results of the thesis show that parameter-driven AFT generation can support early architecture-phase safety-security reasoning. The framework helps the analyst identify which attack-fault paths are structurally applicable to a given architecture and how those paths change when architecture parameters are modified. It also supports a semi-quantitative comparison of dominant paths, which can help highlight where further analysis, mitigation, or verification effort should be focused.

Future work can extend the framework in several directions. A first direction is extending the framework artefacts to additional systems and architecture contexts. The same process could be reproduced with a different parameter schema, fragment library, and activation rule set to describe systems other than multi-tenant onboard software architectures within SataaS missions. Even within the use case implemented in this thesis, additional architecture parameters could refine the

system description and reveal attack-fault paths not covered by the current artefacts.

A second direction is expanding the fragment library to additional top events. This thesis implements the *Tenant-Platform Privilege Boundary Enforcement Failure* case, but other candidates include tenant-tenant non-interference failure, spacecraft-interface command boundary failure, and software execution integrity failure. Each top event would require its own BFMs, FEs, AE-FC pairs, CWE mappings, and activation rules, while still following the same metamodel and overall fragment-activation rationale.

A third direction is formalisation. The current framework defines its artefacts and process at the level of method description and prototype implementation. A stricter formal definition of the architecture model, fragment structure, activation semantics, and aggregation rules would make the framework easier to validate, extend, and reproduce. It would also support stronger claims about consistency between the encoded artefacts and the generated AFT outputs.

A fourth direction is refinement of the likelihood-analysis method. The current implementation uses an ordinal 1–5 scale and propagates OR-gate likelihood using the maximum value. This is suitable for identifying the most accessible path, but it can saturate when the top event reaches the highest likelihood category. In such cases, two variants may both receive a Very High top-event likelihood even if one variant has a broader set of high-likelihood paths. Future work could therefore add secondary comparison indicators, such as the number of BFMs and FEs reaching High or Very High likelihood, the distribution of likelihood values across the tree, or a separate exposure-profile metric. This would allow variants with the same top-event likelihood to be compared in more detail.

A fifth direction is integration of additional security knowledge sources or databases beyond CWE. Examples include SPARTA, which could support space-specific adversarial technique modelling, and CAPEC, which could support attack-pattern reasoning. These sources would need to be mapped carefully because they correspond to different lifecycle phases, abstraction levels, and modelling purposes.

A sixth direction is further development of the prototype implementation. Future versions could provide stronger validation, clearer reporting, improved visualisation of large generated trees, and a more integrated workflow for generation, comparison, and likelihood analysis. Broader evaluation using additional architecture variants, selected top events, and expert-reviewed case studies would also help assess the robustness of the parameter schema, fragment library, and activation rules across a wider design space.

Overall, this thesis showed that parameter-driven AFT generation can support early reasoning about safety-security propagation in multi-tenant onboard software. By linking design choices to activated attack-fault paths, the proposed framework makes architecture comparison more traceable, repeatable, and easier to interpret during preliminary design.

References

- [1] P. Lin, K. Abney, B. DeBruhl, K. Abercromby, H. Danielson, and R. Jenkins, "Outer space cyberattacks: Generating novel scenarios to avoid surprise," Ethics + Emerging Sciences Group, California Polytechnic State University, Tech. Rep. Version 1.0.1, Jun. 2024. [Online]. Available: <https://ethics.calpoly.edu/spacecyberattacks.pdf>
- [2] C. Swope, K. A. Bingen, M. Young, and K. LaFave, "Space threat assessment 2025," Center for Strategic and International Studies (CSIS), Apr. 2025. [Online]. Available: <https://www.csis.org/analysis/space-threat-assessment-2025>
- [3] Z. A. H. Hammadeh et al., "Designing secure space systems," in *2025 Design, Automation and Test in Europe Conference (DATE)*, IEEE, 2025. DOI: 10.23919/DATE64628.2025.10992767
- [4] G. Kavallieratos and S. Katsikas, "An exploratory analysis of the last frontier: A systematic literature review of cybersecurity in space," *International Journal of Critical Infrastructure Protection*, vol. 43, p. 100640, 2023, ISSN: 1874-5482. DOI: <https://doi.org/10.1016/j.ijcip.2023.100640>
- [5] J. Park, T. Eom, H. Kim, H. Park, Z. Yoon, and J. Park, "Threat Vector–Hierarchical Attack Representation Model-Based Threat Modeling and Security Assessment for Satellite Networks," *Applied Sciences*, vol. 15, no. 5, 2025, ISSN: 2076-3417. DOI: 10.3390/app15052751
- [6] H. Otte et al., "Preliminary design of the stellar apps software platform for developing and executing on-board applications," in *Proceedings of the 2025 European Data Handling and Data Processing Conference for Space, EDHPC 2025*, Jan. 2026, ISBN: 979-8-3503-7954-9. [Online]. Available: <https://elib.dlr.de/222271/>
- [7] MITRE Corporation, *CWE-770: Allocation of Resources Without Limits or Throttling*, 2026. [Online]. Available: <https://cwe.mitre.org/data/definitions/770.html>
- [8] "ECSS-E-ST-40C Rev.1: Space Engineering – Software," ECSS Secretariat, ESA-ESTEC, Requirements & Standards Section, Noordwijk, The Netherlands, ECSS Standard ECSS-E-ST-40C Rev.1, Apr. 2025.
- [9] A. Motii, "Mbta: A model-based threat analysis approach for software architectures," in *Computer Safety, Reliability, and Security*, J. Guiochet, S. Tonetta, and F. Bitsch, Eds., Cham: Springer Nature Switzerland, 2023, pp. 121–134.

- [10] S. Pinchinat, M. Acher, and D. Vojtisek, "ATSyRa: An Integrated Environment for Synthesizing Attack Trees," in *Second International Workshop on Graphical Models for Security (GraMSec'15) co-located with CSF'15*, Verona, Italy, Jul. 2015. [Online]. Available: <https://inria.hal.science/hal-01243021>
- [11] D. Marszk, D. Fischer, N. R. Carvalho, T. Oerther, M. Henkel, and G. Labrèche, "Innovating Ground Software Systems – A Retrospective of OPS-SAT-1 Mission," in *Proceedings of the 18th International Conference on Space Operations*, ser. SpaceOps 2025, SpaceOps-2025, ID 281, Montreal, Canada, May 2025, pp. 1–12.
- [12] DPhi Space, *Software Payload Hosting: Deploy your code in space*, n.d. [Online]. Available: <https://software.dphispace.com/>
- [13] J. Willbold, M. Schloegel, M. Vögele, M. Gerhardt, T. Holz, and A. Abbasi, "Space odyssey: An experimental software security analysis of satellites," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 1–19. DOI: 10.1109/SP46215.2023.10351029
- [14] Docker.com, *How Exodus Orbitals Simplifies Satellite and Space Prototyping with Docker*, n.d. [Online]. Available: <https://www.docker.com/customer-stories/exodus-orbitals/>
- [15] R. Jia, Y. Yang, J. Grundy, J. Keung, and L. Hao, "A systematic review of scheduling approaches on multi-tenancy cloud platforms," *Information and Software Technology*, vol. 132, p. 106478, Apr. 2021. DOI: 10.1016/j.infsof.2020.106478
- [16] S. Severin et al., "ROSPIN-SAT-1: Romania's First Open Source Earth Observation CubeSat Mission," in *Proceedings of the 75th International Astronautical Congress*, ser. IAC 2024, IAC-24-B4.4.2.x83580, Milan, Italy, Oct. 2024, pp. 1–15.
- [17] R. Wenker et al., "On-board software architecture in MTG satellite," in *2017 IEEE International Workshop on Metrology for AeroSpace (MetroAeroSpace)*, 2017, pp. 318–323. DOI: 10.1109/MetroAeroSpace.2017.7999588
- [18] D. McComas, J. Wilmot, and A. Cudmore, "The Core Flight System (cFS) Community: Providing Low Cost Solutions for Small Spacecraft," in *Proceedings of the 30th Annual AIAA/USU Conference on Small Satellites*, Paper SSC16-IV-1, NASA Document ID 20160010300, Report No. GSFC-E-DAA-TN33786, Salt Lake City, UT, USA, Aug. 2016.
- [19] C. Coelho, O. Koudelka, and M. Merri, "Nanosat mo framework: When obsw turns into apps," in *2017 IEEE Aerospace Conference*, 2017, pp. 1–8. DOI: 10.1109/AERO.2017.7943951
- [20] Y. Yao, L. Chang, X. Yu, and H. Zhang, "Application-oriented on-board software management system for micro-nano satellite," in *IGARSS 2024 - 2024 IEEE International Geoscience and Remote Sensing Symposium*, 2024, pp. 6900–6903. DOI: 10.1109/IGARSS53475.2024.10642266
- [21] C. Wang, Y. Zhang, Q. Li, A. Zhou, and S. Wang, "Satellite computing: A case study of cloud-native satellites," in *2023 IEEE International Conference on Edge Computing and Communications (EDGE)*, 2023, pp. 262–270. DOI: 10.1109/EDGE60047.2023.00048

- [22] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The Architecture Analysis & Design Language (AADL): An Introduction," Carnegie Mellon University, Software Engineering Institute, Technical Note CMU/SEI-2006-TN-011, Feb. 2006, Accessed: 1 June 2026. DOI: 10.1184/R1/6584909.v1
- [23] A. Mavridou, E. Stachtiari, S. Bliudze, A. Ivanov, P. Katsaros, and J. Sifakis, "Architecture-Based Design: A Satellite On-Board Software Case Study," École Polytechnique Fédérale de Lausanne, Technical Report EPFL-REPORT-221156, 2016. [Online]. Available: <https://infoscience.epfl.ch/entities/publication/ed383011-8992-41c2-93d6-cc14693f9434>
- [24] E. Amoroso, T. Nguyen, J. Weiss, J. Watson, P. Lapiska, and T. Starr, "Toward an approach to measuring software trust," in *Proceedings. 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, 1991, pp. 198–218. DOI: 10.1109/RISP.1991.130788
- [25] "ECSS-Q-HB-80-03A Rev.1: Space Product Assurance – Software Dependability and Safety," ECSS Secretariat, ESA-ESTEC, Requirements & Standards Division, Noordwijk, The Netherlands, ECSS Handbook ECSS-Q-HB-80-03A Rev.1, Nov. 2017.
- [26] S. C. Jero, J. M. Furgala, M. A. Heller, B. R. Nahill, S. A. Mergendahl, and R. W. Skowrya, "Securing the satellite software stack," in *Workshop on Security of Space and Satellite Systems (SpaceSec)*, San Diego, CA, USA, Mar. 1, 2024.
- [27] 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)*, 2024, pp. 1–11. DOI: 10.23919/3S60530.2024.10592292
- [28] F.-X. Molina, E. Baccelli, K. Zandberg, D. Donsez, and O. Alphand, "Cubedate: Securing software updates in orbit for low-power payloads hosted on cubesats," in *2023 12th IFIP/IEEE International Conference on Performance Evaluation and Modeling in Wired and Wireless Networks (PEMWN)*, 2023, pp. 1–6. DOI: 10.23919/PEMWN58813.2023.10304910
- [29] G. Marra, U. Planta, P. Wüstenberg, and A. Abbasi, "On the feasibility of cubesats application sandboxing for space missions," in *Proceedings 2024 Workshop on Security of Space and Satellite Systems*, ser. SpaceSec 2024, Internet Society, 2024. DOI: 10.14722/spacesec.2024.23033 [Online]. Available: <http://dx.doi.org/10.14722/spacesec.2024.23033>
- [30] R. Kumar and M. Stoelinga, "Quantitative security and safety analysis with attack-fault trees," Jan. 2017. DOI: 10.1109/HASE.2017.12
- [31] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, "Fault tree handbook," U.S. Nuclear Regulatory Commission, Washington, DC, Tech. Rep. NUREG-0492, 1981, Manuscript completed March 1980.
- [32] M. M. Hugue, "Fault type enumeration and classification," Office of Naval Research, Tech. Rep. ONR-910915-MCM-TR9105, Nov. 1991, Available through Defense Technical Information Center, DTIC accession number AD-A242 541.
- [33] S. Nikdel and J. Shortle, "Identifying critical nodes in fault tree safety models with limited data," in *2024 Integrated Communications, Navigation and Surveillance Conference (ICNS)*, 2024, pp. 1–9. DOI: 10.1109/ICNS60906.2024.10550693

- [34] E. van Breukelen, R. Hamann, and E. Overbosch, "Qualitative fault tree analysis applied as a design tool in a low cost satellite design: Method and lessons learned," Apr. 2006.
- [35] S. Jung, J. Yoo, and Y.-J. Lee, "A software fault tree analysis technique for formal requirement specifications of nuclear reactor protection systems," *Reliability Engineering & System Safety*, vol. 203, p. 107 064, 2020, ISSN: 0951-8320. DOI: <https://doi.org/10.1016/j.ress.2020.107064>
- [36] B. Kordy, L. Piètre-Cambacédès, and P. Schweitzer, "DAG-based attack and defense modeling: Don't miss the forest for the attack trees," *Computer Science Review*, vol. 13-14, pp. 1–38, 2014, ISSN: 1574-0137. DOI: <https://doi.org/10.1016/j.cosrev.2014.07.001>
- [37] W. Wideł, M. Audinot, B. Fila, and S. Pinchinat, "Beyond 2014: Formal Methods for Attack Tree-based Security Modeling," *ACM Comput. Surv.*, vol. 52, no. 4, Aug. 2019, ISSN: 0360-0300. DOI: 10.1145/3331524
- [38] 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)*, 2021, pp. 68–76. DOI: 10.1109/SMC-IT51442.2021.00016
- [39] N. F. I, M. M, and D. A, "Integrating Cyber Attacks within Fault Trees," *RELIABILITY ENGINEERING & SYSTEM SAFETY*, vol. 94, no. 9, pp. 1394–1402, 2009, ISSN: 0951-8320. DOI: 10.1016/j.ress.2009.02.020
- [40] M. Lopuhaä-Zwakenberg, *Quantitative analysis of attack-fault trees via markov decision processes*, Aug. 2024. DOI: 10.48550/arXiv.2408.06914
- [41] A.-M. Konsta, A. Lluch Lafuente, B. Spiga, and N. Dragoni, "Survey: Automatic generation of attack trees and attack graphs," *Computers & Security*, vol. 137, p. 103 602, 2024, ISSN: 0167-4048. DOI: <https://doi.org/10.1016/j.cose.2023.103602>
- [42] R. Vigo, F. Nielson, and H. R. Nielson, "Automated generation of attack trees," in *2014 IEEE 27th Computer Security Foundations Symposium*, 2014, pp. 337–350. DOI: 10.1109/CSF.2014.31
- [43] J. Bryans, L. S. Liew, H. N. Nguyen, G. Sabaliauskaite, S. Shaikh, and F. Zhou, "A template-based method for the generation of attack trees," in *Information Security Theory and Practice*, M. Laurent and T. Giannetsos, Eds., Cham: Springer International Publishing, 2020, pp. 155–165, ISBN: 978-3-030-41702-4.
- [44] R. Jhawar, K. Lounis, S. Mauw, and Y. Ramírez-Cruz, "Semi-automatically augmenting attack trees using an annotated attack tree library," in *Security and Trust Management*, S. K. Katsikas and C. Alcaraz, Eds., Cham: Springer International Publishing, 2018, pp. 85–101, ISBN: 978-3-030-01141-3.
- [45] CVE Program, *CVE and NVD Relationship*, Page last updated or reviewed: 11 December 2020. Accessed: 2026-06-01, 2020. [Online]. Available: https://www.cve.org/Resources/Media/Archives/OldWebsite/about/cve_and_nvd_relationship.html
- [46] I. Pekaric et al., "Streamlining attack tree generation: A fragment-based approach," in *57th Annual Hawaii International Conference on System Sciences, HICSS 2024*, 2024, pp. 7447–7456, ISBN: 978-099813317-1. [Online]. Available: <https://elib.dlr.de/211379/>

- [47] MITRE, *Common Weakness Enumeration (CWE)*. [Online]. Available: <https://cwe.mitre.org/index.html>
- [48] ECSS-E-ST-80C: *Space Engineering - Security in Space Systems Lifecycles*, European Cooperation for Space Standardization, Jul. 1, 2024.
- [49] “Technical Guideline BSI TR-03184: Information Security for Space Systems, Part 1: Space Segment,” Federal Office for Information Security, Technical Guideline BSI TR-03184, version 1.0, Jul. 28, 2023.
- [50] The Aerospace Corporation, *Space Attack Research & Tactic Analysis (SPARTA)*, 2022. [Online]. Available: <https://sparta.aerospace.org/>
- [51] “Security Threats Against Space Missions,” Consultative Committee for Space Data Systems, Washington, DC, USA, Informational Report, Green Book CCSDS 350.1-G-3, Feb. 2022, Published and maintained by the CCSDS Secretariat, National Aeronautics and Space Administration.
- [52] CVE Program, *Common Vulnerabilities and Exposures (CVE)*, 2026. [Online]. Available: <https://www.cve.org/>
- [53] MITRE, *Common Attack Pattern Enumeration and Classification (CAPEC)*. [Online]. Available: <https://capec.mitre.org/>
- [54] A. T. Sheik, U. I. Atmaca, C. Maple, and G. Epiphaniou, “Challenges in threat modelling of new space systems: A teleoperation use-case,” *Advances in Space Research*, vol. 70, no. 8, pp. 2208–2226, 2022, ISSN: 0273-1177. DOI: <https://doi.org/10.1016/j.asr.2022.07.013>
- [55] “ECSS-Q-ST-80C Rev.2: Space Product Assurance – Software Product Assurance,” ECSS Secretariat, ESA-ESTEC, Requirements & Standards Section, Noordwijk, The Netherlands, ECSS Standard ECSS-Q-ST-80C Rev.2, Apr. 2025.
- [56] OWASP Foundation, *A02:2025 security misconfiguration - owasp top 10:2025*, 2025. [Online]. Available: https://owasp.org/Top10/2025/A02_2025-Security_Misconfiguration/