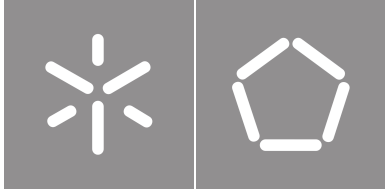
Universidade do Minho
Escola de Engenharia

José Miguel Alves Pires

Trustworthy Open-Source Reference Software Stack for UAV applications

July, 2025

Trustworthy Open-Source Reference Software Stack for UAV applications
Jose Pires
UMinho | 2025



Universidade do Minho
Escola de Engenharia

José Miguel Alves Pires

Trustworthy Open-Source Reference Software Stack for UAV applications

Master Thesis
Master in Industrial Electronics and Computers Engineering
Specialization in Embedded Systems

Work developed under the supervision of:

Professor Doctor Sandro Pinto

July, 2025

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.2.5) [1].

Acknowledgements

First and foremost, I would like to thank my supervisor, Prof. Sandro Pinto, for all the guidance, support, and technical expertise that was so valuable throughout this process.

To my parents for always pushing me forward and for being the life example of character and sheer determination, a lamp lit on a stormy day. Thank you also for an upbringing full of debates and argumentation, which has undoubtedly contributed for mastering the challenges I took ahead, and for instilling in me the taste in good music which helped me throughout this journey, listening to the *Music of the Spheres*, and sorry for my constant lack of time. To my sister, for always being so supportive and caring, helping out with your immense ingenuity and creativity, a mix between the engineer and the artist.

To my closest friends, Pedro and Chicão for taking me out to dinner, which is exactly what one needs sometimes. For all the support and the true friendship that survives even the end of times. To my beloved dog Pantufa for adopting me eleven years ago when I was going crazy with my first MSc thesis and never leaving my side.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

_____, _____

(Place) (Date)

_____

(José Miguel Alves Pires)

”

“Wir müssen wissen, Wir werden wissen.
We must know, We shall know.”

— David Hilbert, 1930
(mathematician)

Resumo

Pilha de software de referência para aplicações UAV

O crescente mercado de Unmanned Aerial Vehicles (UAVs) apresenta desafios significativos em termos de segurança operacional e contra ameaças, frequentemente negligenciados nos projetos atuais de sistemas. As aplicações de UAVs possuem inerentemente restrições de tempo real com níveis mistos de criticidade, significando que o risco de falha varia em severidade. Por exemplo, uma falha na vigilância por vídeo, função não crítica, poderia propagar-se para o controlador de voo crítico, levando a consequências catastróficas como uma queda e danos a pessoas ou bens. As pilhas de voo tradicionais multiplataforma para UAVs, embora abordem a criticidade mista ao separar funções em diferentes nós de hardware, aumentam indesejavelmente o peso e a dimensão do UAV e frequentemente carecem de garantias de isolamento. Adicionalmente, as soluções open-source existentes geralmente não atendem plenamente aos requisitos de criticidade mista e segurança, enquanto as alternativas comerciais frequentemente carecem de transparência. Para enfrentar estas questões críticas, este trabalho propõe o Supervised Single-Platform Flight Stack (SSPFS), uma abordagem inovadora que integra as funções do controlador de voo e do computador companheiro numa única plataforma Raspberry Pi 4 + PilotPi shield sob o hipervisor Bao. Esta consolidação é alcançada mediante o hipervisor Bao, um hipervisor bare-metal leve, orientado para segurança operacional e contra ameaças, projetado para sistemas embebidos de tempo real e de criticidade mista. O Bao foi estendido com um supervisor de mailbox personalizado para mediar transações de firmware através de um barramento partilhado. O piloto automático PX4 foi selecionado devido à sua natureza open-source, amplo suporte a plataformas, arquitetura modular e ampla adoção na indústria. Uma contraparte Unsupervised Single-Platform Flight Stack (USPFS) também foi desenvolvida para quantificar os benefícios da camada de supervisão. Resultados experimentais demonstram que o Supervised Single-Platform Flight Stack (SSPFS) fornece forte isolamento: falhas injetadas na Virtual Machine (VM) não crítica (Companion) levam à sua falha, enquanto a VM crítica (PX4) mantém funcionalidade total, mantendo o UAV no ar. Benchmarking com o MiBench mostra uma pequena degradação de desempenho, e o particionamento melhorado da cache reduz interferências. A sobrecarga de escalonamento em worst-case para tarefas do PX4 foi muito baixa (2%). A taxa de captura da câmara na Companion VM também não apresentou sobrecarga estatisticamente significativa (worst-case de 2%). Em alguns casos, o SSPFS até superou o sistema USPFS, atribuído ao particionamento estático de recursos. Testes de voo real confirmam acompanhamento preciso de posição e uma modesta sobrecarga de

CPU (≈6% média), validando que o SSPFS atende aos requisitos de tempo real sem sacrificar segurança. Este estudo estabelece um caminho viável para sistemas consolidados de criticidade mista para UAVs confiáveis, com impacto mínimo no desempenho.

Palavras-chave: UAV, Sistemas de Criticidade mista, Hipervisor, Bao, Segurança, PX4

Abstract

Trustworthy Open-Source Reference Software Stack for UAV applications

The booming market for UAVs brings forth significant challenges concerning security and safety, often overlooked in current system designs. UAV applications inherently possess real-time constraints with mixed-criticality levels, meaning the risk of failure varies in severity. For instance, a failure in video surveillance, a non-critical function, could propagate to the critical flight controller, leading to catastrophic consequences like a crash and harm to people or goods. Traditional multi-platform flight stacks for UAVs, while addressing mixed-criticality by separating functions onto different hardware nodes, undesirably increase the UAV's weight and footprint and often lack isolation guarantees. Moreover, existing open-source solutions often fall short in meeting mixed-criticality and security requirements, while commercial alternatives frequently lack transparency. To address these critical issues, this work proposes the Supervised Single-Platform Flight Stack (SSPFS), an innovative approach that integrates the flight controller and companion computer functions on a single Raspberry Pi 4 + PilotPi shield platform under the Bao hypervisor. This consolidation is achieved by leveraging the Bao hypervisor, a lightweight, security, and safety-oriented bare-metal hypervisor designed for embedded real-time systems and mixed-criticality systems. Bao was extended with a custom mailbox supervisor to mediate firmware transactions over a shared bus. The PX4 autopilot was selected due to its open-source nature, extensive platform support, modular architecture, and widespread adoption in the industry. An USPFS counterpart was also developed to quantify the benefits of the supervision layer. Experimental results demonstrate that SSPFS provides strong isolation: injected faults in the non-critical Companion VM lead to its failure, while the critical PX4 VM maintains full functionality, keeping the UAV airborne. Benchmarking with MiBench shows a small performance degradation and improved cache partitioning reduces interference. The worst-case scheduling overhead for PX4 tasks was very low (2%). The camera's frame rate in the Companion VM also showed no statistically significant overhead (worst-case of 2%). In some instances, the SSPFS even outperformed the USPFS system, attributed to the static partitioning of resources. Real-flight tests confirm accurate position tracking and a modest CPU overhead (≈6 % average), validating that SSPFS meets real-time requirements without sacrificing safety. This study establishes a viable path toward trustworthy, consolidated mixed-criticality UAV systems with minimal performance impact.

Keywords: UAV, Mixed-Criticality Systems, Hypervisor, Bao, Security, PX4.

Contents

Appendices

List of Figures

List of Listings

Acronyms

ABI Application Binary Interface *(p. 32)*
ADC Analog to Digital Converter *(p. 25)*
AI Artificial Intelligence *(pp. 28, 29, 37)*
AICS Automotive and Industrial Control Suite *(pp. 3, 72, 80, 92)*
API Application Programming Interface *(pp. 17, 31, 33, 38, 42)*
ASIL Automotive Safety and Integrity Level *(p. 5)*
ATF Arm Trusted Firmware *(pp. 58, 67, 70)*

BLDC Brushless Direct Current *(pp. 17, 20, 46, 47)*
BOM Bill Of Materials *(p. 22)*
BSD Berkeley Software Distribution *(pp. 22, 26, 27, 31, 33, 37)*
BSP Board Support Package *(pp. 58, 66)*
BVLOS Beyond Visual Line-Of-Sight *(p. 13)*

CAD Computer-Aided Design *(p. 19)*
CAN Controller Area Network *(pp. 22, 25–27)*
CMA Contiguous Memory Allocation *(p. 52)*
COTS Commercial Off-The-Shelf *(p. 22)*
CPU Central Processing Unit *(pp. 6, 7, 9, 10, 27, 45, 48–52, 55, 58, 59, 61, 84, 90, 94, 100)*
CSI Camera Serial Interface *(pp. 25, 26, 48)*
CSP Context Security Policy *(p. 31)*

DAL Design/Development Assurance Level *(p. 5)*
DC Direct Current *(pp. 17, 60)*
DDoS Distributed Denial Of Service *(p. 18)*
DIY Do It Yourself *(p. 46)*

DMA Direct Memory Access *(pp. 49, 52, 68, 85, 101)*
DoS Denial of Service *(pp. 10, 18)*
DTB Device Tree Blob *(pp. 58, 59)*
DTB Device Tree Source *(p. 67)*

EEPROM Electrically Erasable Programmable Read-Only Memory *(p. 32)*
eMMC embedded Multi Media Card *(p. 27)*
ESC Electronic Speed Controller *(pp. 17, 46, 47, 60, 63)*

FAA Federal Aviation Administration *(pp. 13, 17)*
FCS Flight Control System *(pp. 17, 20, 22, 29, 32, 35, 36)*
FMU Flight Management Unit *(pp. 22, 25–28, 33, 37, 38, 42–44, 46, 47, 53, 61, 83, 88, 90–92, 94, 100)*
FPGA Field-Programmable Gate Array *(pp. 23, 25, 37)*
FPS Frames Per Second *(pp. 52, 64, 84, 85)*
FPV First-Person View *(pp. 18, 19, 35)*

GCS Ground Control Station *(pp. 16–19, 21, 26, 29, 31–33, 35–37, 40–43, 47, 60, 62, 64, 65, 80, 84)*
GIC Generic Interrupt Controller *(pp. 11, 58, 67)*
GNSS Global Navigation Satellite System *(pp. 16, 17, 25)*
GPIO General Purpose Input/Output *(p. 49)*
GPL GNU Public License *(pp. 22, 32)*
GPOS General-Purpose Operating System *(pp. 31, 38, 42, 43, 101)*
GPS Global Positioning System *(pp. 16–18, 21, 22, 25, 28, 29, 32, 37, 46, 48, 49, 60, 89)*
GPU Graphics Processing Unit *(pp. 27, 48–51, 55)*
GSI Generic Serial Interface *(p. 25)*
GUI Graphical User Interface *(pp. 88, 90, 91)*

HAL Hardware Abstraction Layer *(pp. 31, 32, 38)*
HAP High Altitude Platform *(p. 14)*
HD High-Definition *(p. 52)*
HDL Hardware Description Language *(p. 22)*
HFC Hydrogen Fuel Cell *(p. 14)*
HW Hardware *(pp. 2, 6–11, 16, 18–20, 22, 23, 25, 29, 31, 38)*

I/O Input/Output *(pp. 8, 10, 11, 20–22, 25, 27, 43)*

I2C Inter-Integrated Circuit *(pp. 22, 24–27, 58, 65)*
IMU Inertial Measuring Unit *(pp. 16, 17, 20, 22, 24–26, 37, 49)*
IOMMU Input/Output Memory Management Unit *(p. 8)*
IP Internet Protocol *(pp. 37, 60, 64)*
ISA Instruction Set Architecture *(p. 10)*

LAN Local Area Network *(p. 42)*
LAP Low Altitude Platform *(pp. 14, 45)*
LiDAR Light Detection And Ranging *(pp. 16, 17, 19, 29)*
LiPo Lithium Polymer *(pp. 14, 46, 47, 60)*
LLC Last-Level Cache *(pp. 10, 11, 80, 101)*
LOS Line-Of-Sight *(p. 21)*
LTE Long-Term Evolution *(pp. 26, 27, 37)*

Mbps Megabits per second *(p. 52)*
MCS Mixed-Criticality System *(pp. 1, 2, 5–7, 9, 10, 37, 78)*
MCU Micro Controller Unit *(pp. 22, 24, 46)*
MMU Memory Management Unit *(p. 8)*

NASA National Aeronautics and Space Administration *(p. 13)*
NPU Neural Processing Unit *(p. 28)*

OOM Out Of Memory *(pp. 75, 77)*
OS Operating System *(pp. 6–10, 18, 22, 28, 30–33, 35–38, 44, 45, 48, 52, 56, 60, 65, 67, 70, 72)*
OSH Open-Source Hardware *(pp. 22, 32)*
OSS Open-Source Software *(pp. 31–33)*
OTA Over-The-Air *(p. 34)*

PCB Printed Circuit Board *(pp. 22, 47)*
PCIe Peripheral Component Interconnect Express *(pp. 68, 99)*
PID Proportional Integrative Derivative *(p. 20)*
PMU Performance Monitor Unit *(p. 80)*
PV Photovoltaic *(p. 14)*
PWM Pulse-Width Modulation *(pp. 22, 25–27, 47, 48, 63, 65)*

QoS Quality of Service *(p. 7)*

RAM Random Access Memory *(pp. 22, 25, 27, 45, 52, 90, 94, 101)*

RC Radio Controller *(pp. 17, 20, 21, 25, 31, 33, 37, 42, 46–49, 61, 65)*

RID Remote IDentifier *(p. 17)*

ROS Robotic Operating System *(pp. 18, 31)*

RT Real-Time *(pp. 6, 7)*

RTC Real-Time Communication *(p. 26)*

RTOS Real-Time Operating System *(pp. 31, 33–36, 38, 42, 43, 47, 60, 101)*

RTPS Real-Time Publish Subscribe *(p. 31)*

RTSP Real-Time Streaming Protocol *(pp. 64, 65)*

SD Storage Disk *(pp. 25, 27, 47, 48, 58)*

SDK Software Development Kit *(pp. 14, 17, 28, 31, 33, 38, 42)*

SDR Software-Defined Radio *(p. 18)*

SIL Safety Integrity Level *(p. 5)*

SoC System on Chip *(pp. 25, 48)*

SPI Serial Peripheral Interface *(pp. 22, 24–27, 52, 58, 65)*

SRAM Static Random Access Memory *(pp. 22, 46)*

SSH Secure Shell *(pp. 88, 90, 91)*

SSPFS Supervised Single-Platform Flight Stack *(pp. vi–viii, 2–4, 39, 44, 48, 49, 52–54, 56, 58–60, 67, 70, 72, 74–78, 80, 84, 86, 88, 92–94, 99, 101)*

SW Software *(pp. 7, 10, 16, 17, 19, 20, 22, 24, 25, 29, 31, 32, 34, 35, 38, 42)*

SWaP-C Size, Weight, Power and Cost *(pp. 1, 2, 5, 6, 38, 39)*

SWD Serial Wire Debug *(p. 47)*

TCB Trusted Computing Base *(pp. 10, 11)*

TEE Trusted Execution Environment *(p. 10)*

TLB Translation Lookaside Buffer *(pp. 11, 80)*

TOF Time-of-Flight *(p. 17)*

TOPS Trillion Operations Per Second *(p. 28)*

UART Universal Asynchronous Receiver-Transmitter *(pp. 22, 24–27, 42, 47, 49, 58, 60, 65, 67, 80)*

UAS Unmanned Aerial System *(pp. 1, 11)*

UAV Unmanned Aerial Vehicle *(pp. vi–viii, 1–6, 11–20, 25, 29, 31, 32, 34–38, 40, 42, 45–47, 52, 56, 60, 62, 64, 65, 73, 83, 84, 88, 90–92, 99, 113)*

UAVIC Unmanned Aerial Vehicle Integrated Controller *(pp. 3, 4, 43–45, 47–49, 54, 56, 58, 60, 65, 67, 72, 80, 99–101)*

UDP User Datagram Protocol *(pp. 26, 42, 64, 65, 84)*

UGV Unmanned Ground Vehicle *(p. 11)*
UI User Interface *(p. 17)*
UMPFS Unsupervised Multi-Platform Flight Stack *(pp. 3, 39, 41)*
uORB Micro Object Request Broker *(pp. 31, 88, 90)*
USB Universal Serial Bus *(pp. 22, 24–26, 28, 48, 49, 52, 54, 60, 65, 68, 88)*
USPFS Unsupervised Single-Platform Flight Stack *(pp. vi, viii, 3, 39, 42, 43, 49, 50, 52, 53, 56, 58–60, 65, 67, 70, 72, 74–78, 80, 84, 86, 88, 91, 92, 99)*
USV Unmanned Surface Vehicle *(p. 11)*
UTM UAS Traffic Management *(p. 13)*
UUV Unmanned Under water Vehicle *(p. 11)*
UV Unmanned Vehicle *(pp. 11, 31, 32)*

vCPU Virtual CPU *(p. 80)*
VM Virtual Machine *(pp. vi, viii, 3, 6–11, 36, 44, 45, 49, 52, 53, 55, 56, 59, 73, 84, 88, 90, 91, 93, 94, 100, 101)*
VMA Virtual Memory Area *(p. 77)*
VMM Virtual Machine Monitor *(p. 7)*
VPN Virtual Private Network *(p. 26)*
VTOL Vertical Take-Off and Landing *(pp. 13, 28, 45)*

WCET Worst-Case Execution Time *(p. 6)*

XML eXtensible Markup Language *(p. 32)*

1 Introduction

"*The best way to predict the future is to invent it.*"
– Alan Kay, computer scientist

Unmanned Aerial Vehicles (UAVs), Unmanned Aerial Systems (UASs), or more commonly drones, are a class of unmanned robotic vehicles that can execute flying missions and carry payloads, guided either by remote control stations or in an autonomous way [2, 3].

The Unmanned Aerial Vehicle (UAV)s market is booming. In 2017 the North America market had a revenue of 737 million USD dollars and a eight-fold increase is expected for 2026 with a staggering 6.7 billion USD dollars, with strong contributions from the sectors of agriculture and farming, and security and law enforcement [4].

The versatility and utility of UAVs is evident in its wide range of applications, performing tasks with high added value and that would be somewhat hard or impossible for a person to achieve: rescue operations and saving lifes, agriculture and farming, building structures, pipeline inspections, delivering goods and medical supplies, video capturing and filming, surveying, inventory management, providing telecommunications in remote areas, among others [2].

However, only recently regulations have been explicitly enforced on UAVs, with many countries allowing drones to fly over populated areas (at altitudes lower than 150 m) [5]. This poses several safety and security concerns. First, because a drone's crash, unintentional or not, can severely harm people and goods. Secondly, because they are cyber-physical systems which are able to record sensitive and private information. Thus, it is critical to ensure the security of an UAV system.

Furthermore, UAV's applications have real-time constraints but with mixed criticality levels, i.e., the risk of failure is more severe in some case than others. For example, flight control (safety-critical) and video streaming (non-critical) coexist. A compromise in video software could cascade to flight systems, causing catastrophic failure—yet strict hardware separation increases weight and reduces flight time.

This exemplifies a Mixed-Criticality System (MCS): systems executing tasks of differing criticality (safety) levels on shared hardware. However, the integration of mixed-criticality components onto a common hardware platform is not an easy task: the designer needs to meet the stringent Size, Weight, Power and

Cost (SWaP-C) metrics and safety requirements provided by safety-critical standards, such as those for automotive [6], avionics [7], and railway [8] industries. While MCS challenges are well-studied in automotive/avionics, UAVs face unique constraints: extreme SWaP-C limitations and the need for fail-operational safety in uncontrolled environments. Current solutions either compromise isolation (single-board) or add weight/energy overhead (multi-board).

As a result of this integration complexity increases, requiring higher computational power, and thus, these hardware platforms are migrating from single cores to multi-cores and in the future many-core architectures [9]. This leads to the core challenge: reconciling *partitioning* for safety assurance with *sharing* for efficiency. Consequently, problems arise in the modelling, design, implementation, and verification of the required hardware and software [9]. Prior research diverges into theoretical (scheduling-focused) and practical (partitioning-focused) approaches, but their incompatibility leaves UAV consolidation unresolved [9].

This thesis bridges this gap by proposing the SSPFS: a Bao hypervisor-based architecture consolidating mixed-criticality functions on a single UAV platform. This solutions demonstrated robust isolation (malicious non-critical crashes don't impact flight control), minimal overhead (<2% in PX4 tasks's scheduling and camera's frame rate), and real-flight validation – enabling secure, lightweight UAVs without safety compromises while meeting stringent SWaP-C constraints.

1.1 Goals

The main goal of the present work is the development of a trustworthy open-source software stack for UAV applications, where security and safety are paramount, thus closing the gap between open-source and commercial solutions and contributing for the widespread adoption of secure and safe features.

To this end several main objectives have been outlined:

1. Identify the UAV application requirements and select the target Hardware (HW) platform with several security primitives essential for hypervisor-based isolation (e.g, virtual memory, memory protection, TrustZone, criptographic accelerators, etc.).

2. Design an open-source software stack for UAV applications leveraging the Bao hypervisor.

3. Provide support for autonomous mode (autopilot).

4. Quantify performance overhead across benchmark scenarios

5. Demonstrate provable security advantages over alternatives

6. Validate the devised solution in real-flight conditions meeting SWaP-C constraints

1.2 Document Structure

This thesis is organized into six chapters that systematically address the design, implementation, and validation of a trustworthy UAV software stack:

- Chapter 2 – **Background and Related Work**: Presents foundational concepts of mixed-criticality systems and virtualization, with a special focus in the hypervisor technology, more specifically the Bao hypervisor. Current UAV hardware/software solutions are analyzed in depth in both the open-source and commercial sphere. The critical gaps in security, safety, and hardware efficiency are identified through comparative assessment of open-source and commercial platforms. Lastly, the related work is presented, showing the current research directions and topics.

- Chapter 3 – **Design**: Proposes the Supervised Single-Platform Flight Stack (SSPFS) architecture leveraging the Bao hypervisor to consolidate flight control and companion functions on a single hardware platform. Contrasts this with unsupervised multi-platform (Unsupervised Multi-Platform Flight Stack (UMPFS)) and single-platform (USPFS) approaches, analyzing trade-offs in weight reduction, isolation guarantees, and security risks. The system architecture was tailored for a video surveillance application based on the PX4 flight stack. Due to the static partitioning nature of the Bao hypervisor, the hardware needed to be selected and mapped across guests. However, as some collisions occurred, a mechanism for mailbox access supervision in the Bao hypervisor was devised. This mechanism enables both VMs to seamlessly communicate with the Unmanned Aerial Vehicle Integrated Controller (UAVIC)'s firmware.

- Chapter 4 – **Implementation**: Details the realization of both USPFS (unsupervised) and SSPFS (supervised) solutions on the UAVIC platform (Raspberry Pi 4 + PilotPi). Covers hardware validation, PX4/video surveillance stack integration, and Bao hypervisor deployment with VM separation. Includes the custom mailbox supervision mechanism for shared device access.

- Chapter 5 – **Evaluation**: Conducts comprehensive functional tests, performance benchmarking (using MiBench Automotive and Industrial Control Suite (AICS)), and real-flight validation. The behavior of the USPFS and SSPFS systems is analyzed when the system is compromised, assessing the isolation effectiveness under malicious attacks. The baseline performance of the USPFS system is established and the performance degradation of the SSPFS system as a consequence of the introduction of the Bao's hypervisor is assessed using the MiBench Automotive and Industrial Control Suite (AICS). Furthermore, the impact of the interference between guests is analyzed, as well as the impact of the mailbox driver patch. The guests are also benchmarked in single- and dual-VM configuration, considering guest-specific metrics, as the PX4 tasks' scheduling overhead and the camera frame rate. Lastly, both systems are evaluated in a real flight scenario, using an automated mission to compare the position tracking and system resource usage across systems. The functional tests are also repeated, but now in real flight.

- Chapter 6 – **Conclusions and Future Work**: Synthesizes findings on the SSPFS's ability to consolidate the UAV's mixed-criticality stacks into a single hardware platform, while providing robust isolation and minimal overhead. Discusses prospects for future work, including the support for different UAVIC platforms, advanced shared resource management, and more extensive and detailed testing, especially in long-duration flights and attack scenarios.
- **Appendices**: Contain supplementary material including: UAV's taxonomy, source code listings, configuration details, and log files.

2

Background and Related Work

"*We stand on the shoulders of giants.*"
– Isaac Newton, mathematician, physicist, astronomer

In this chapter some background is provided concerning mixed criticality systems, its relevance and the challenges it poses, alongside with the current approach to address its complexity. Next, Unmanned Aerial Vehicle (UAV) are discussed, namely the reference hardware and software, both as open-source and commercial solutions. Lastly, the related work is presented, showing the current research directions and topics.

2.1 Mixed Criticality Systems

Embedded systems have stringent requirements both on timing and criticality of the tasks it performs. Criticality designations (e.g. Automotive Safety and Integrity Levels (ASILs), Design/Development Assurance Levels (DALs), and Safety Integrity Levels (SILs)) define the rigor needed to prevent failures, while timing constraints dictate deadline enforcement (hard/soft/firm) [9].

UAVs epitomize this challenge: **flight control (safety-critical)** and **auxiliary functions (e.g. video streaming)** coexist on shared hardware. Without robust isolation a compromise in a non-critical component (.e.g. video malware) can cascade to flight systems. On the other hand, hardware separation increases weight/power, reducing flight time. This necessitates MCSs – systems executing mixed-criticality tasks on consolidated platforms while simultaneously meeting the safety standards (e.g., ISO 26262 [6] or DO-178C [7]) and the stringent SWaP-C constraints.

While MCS principles apply across domains, UAVs face unique hurdles. They must meet **extreme SWaP-C limits**, where every gram needs to be justified, especially in the lower weight classes. UAVs fly in **uncontrolled environments** with fast changing dynamics. As it impossible to account for every erroneous occurrence, it must present fail-operational behavior , i.e. maintaining critical functions during faults. The migration to the mainstream multi-core architectures increases the complexity in resource

sharing [9]. Current approaches force trade-offs: on one hand, multi-board provides high isolation but poor SWaP-C metrics; on the other hand, single-board provides poor isolation but better SWaP-C.

Two major areas of research stem from this dichotomy: a more theorethical one, largely focused on using criticality-specific Worst-Case Execution Times (WCETs) to schedule systems at each criticality level, but at the expenses of high processor utilization; a more practical one, focused on the safe partitioning of a system with the sharing of computational and communication resources, but with increased complexity. Unfortunately, the combination of both areas is not easy: flexible scheduling requires, at least, dynamic partitioning, whereas certified systems require complete separation or, at least, static partitioning [9]. This gap is acute in UAVs where both efficiency and assurance are non-negotiable.

2.1.1 Virtualization

Mixed critical systems require the integration of software components with disparate criticality levels on a shared HW platform. As aforementioned, the most promising approach is the safe partitioning of the system with shared computation resources. This leads to concept of virtualization, a logic abstraction of HW resources with isolation guarantees, using different approaches such as hypervisors, Operating System (OS)-level virtualization or unikernels [10].

Fig. 75 depicts a map of concepts related to the virtualization topic, which can be used as reference through this section. The major distinction between OS- and hypervisor-level virtualization is the former does not provide emulation of the physical HW, where the virtual domain, called container, has its own virtual HW resources (Central Processing Unit (CPU), memory, filesystem, network), and process and user management. This virtual domain acts as an traditional process of an OS, with its own independent resources. OS-level virtualization, or containerization, is mostly used in cloud environments to avoid the replication of the OS stack, further improving application consolidation on the same hardware [10]. Although lacking the HW emulation, this type of virtualization is gaining momentum also in the Real-Time (RT) systems [11, 12] due to some desirable features like tenant isolation, high-availability, fault-tolerance, software migration, and recovery techiques. However, these solutions introduce sources of undeterminism, e.g. dynamic resources assignment, have an extensive code base, and lack security features, difficulting its certification and testing [10].

Another option that is gaining popularity is the unikernel-based virtualization, where it is possible to run a single application in its virtual domain to increase isolation, performance and security. The full software stack of the system (OS components, libraries, language runtime, and application) is compiled into a single VM — unikernel or library OS — which can be run on the physical hardware, or more commonly, on top of a general-purpose hypervisor [10]. Unikernels provide a fast boot time, low memory footprint, high performance, and strong security by leveraging the underlying host hypervisor. However, they lack flexibility as applications need to be manually ported to the underlying unikernel. These solutions require further studies to enable its adoption in MCSs, especially regarding certification and dependability support [10]. Fig 1 illustrates some examples of virtualization approaches: the application, RT and libraries, and the

OS are bundled as a VM and run on top of an hypervisor (hypervisor); a container is formed only with the first two components and run on top of an OS (OS-level virtualization); all components are compiled into a unikernel and run on top of hypervisor (typically) (unikernel-virtualization).

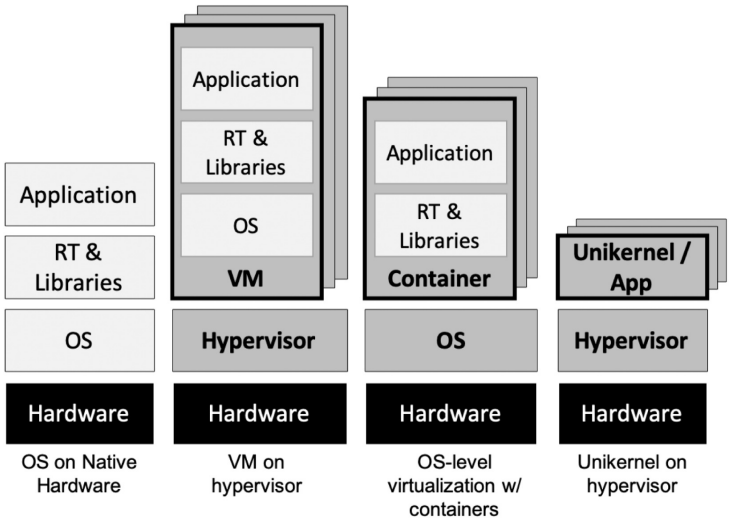


Figure 1: Examples of virtualization approaches [10][1]

Cinque et. al [10] identified the relationship between hypervisor and guest OS combinations, i.e. the type of guest that runs on top of a hypervisor, and the associated applications (Fig. 2). On the left quadrants are represented the guest OSs running on top of a general purpose hypervisor, mainly used for server consolidation in cloud environments (bottom left), and functional testing and prototyping (top left). On the right quadrants are represented the guest OSs on top of a RT hypervisor, mainly used for Quality of Service (QoS) and performance analysis (bottom right), and safety-critical applications (top right). The region of interest – the MCS – is located in the intersection of these right quadrants, emphasizing the need of a RT hypervisor. Thus, the hypervisors will be discussed in more detail in the next section.

2.1.2 Hypervisors

In simple terms a hypervisor is a Virtual Machine Monitor (VMM), a software layer that abstracts the HW resources in order to run distinct and isolated application environments, called VMs or guests, on the same physical machine (see Fig. 1). A VM is the execution environment typically comprised of an OS, or guest OS, and the application Software (SW).

A hypervisor must guarantee isolation in the following domains [10]:

- **Temporal**: ability to isolate or limit the impact of resource consumption (e.g. CPU, network, disk) of a virtual domain on the performance degradation of other virtual domains. Thus, a task from one virtual domain must not cause serious delays to other tasks in other virtual domain.

[1]Used with permission from Elsevier: license nr. 5457890117132
[2]Used with permission from Elsevier: license nr. 5457890117132

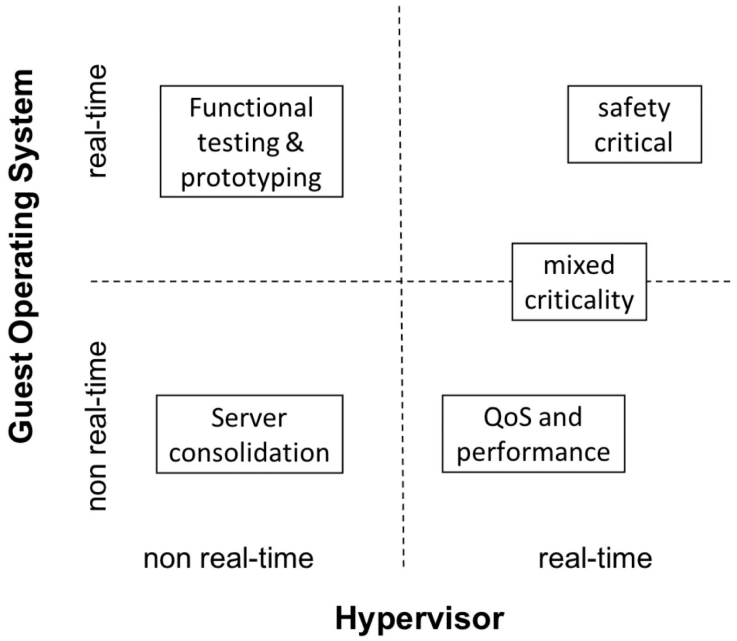


Figure 2: Hypervisor and OS combinations with related applications [10][2]

- **Spatial** (memory-isolation): ability to isolate code and data between virtual domains and between virtual domains and hosts. Thus, a task must not be able to modify private data from other tasks, including devices assigned to a specific task. It is usually implementated using HW memory protection mechanisms (e.g., Memory Management Unit (MMU), Input/Output Memory Management Unit (IOMMU)).

- **Fault**: prevents the propagation of faults from one virtual domain to another that could cause blockages or stop the whole system.

A hypervisor can be classified in numerous ways (see Fig. 75), as follows:

1. **Host presence**: If the hypervisor is executed on top of an existing *host* OS — hosted — is classified as type II, otherwise as type I — bare-metal —, running directly on the HW, acting as a classic OS. The bare-metal hypervisor controls directly the hardware resources (e.g. Jailhouse [13], Bao [14]), whereas the hosted manages it indirectly (e.g. KVM [15], Xen).

2. **Guest awareness of the hypervisor**: A fully virtualized hypervisor abstracts completely the HW resources to the guest, emulating privileged instructions and Input/Output (I/O) operations. Thus, it allows a guest OS or an application to run unmodified, as they were running directly on the physical machine. Typical examples are KVM [15] and Microsoft Hyper-V [16]. Conversely, in a paravirtualized hypervisor (e.g., Xen [17]) the guest OS must communicate with the underlying hypervisor through hypercalls, which is cumbersome, since it requires the guest OS to be modified.

3. **Real-time support**: this refers to the explicit support for the management of the time budget of each VM (e.g. scheduling algorithms) which must meet the explicit timing constraints [10]. These

hypervisors can be further decomposed in dynamic (e.g. KVM[15], Xen [17]) or static (e.g. Jailhouse[13], Bao[14]), depending on the VM resources assignment timing: in run-time, as needed, for the former; in instantiation time for the latter. Static solutions are often employed for MCSs due to higher tolerance to failure and less overhead. Moreover, as they usually have a small code base they are easier to test and certify according to industrial standards [10].

4. **Application type**: embedded, if targeted for a specific application, system, or mission, otherwise they are general-purpose [18].

Solutions based on hypervisors can be grouped in four categories [10]:

1. **Separation kernel and microkernel**: specially designed for industrial and embedded domains. A separation kernel is a special type of a very small bare-metal hypervisor defining fixed VMs and managing information flows, relegating device drivers, user model, shell access and dynamic memory to the guest OS. This simple architecture results in a minimal implementation, ideal for a MCS. Examples are PikeOS [19], Xtratum [20], Jailhouse [13], and Bao [14].

2. **General-purpose**: enhancement of general-purpose hypervisors (e.g. KVM [15], Xen [17]) to support real-time features.

3. **Security CPU HW extensions**: leverage the isolation support provided by these hardware extensions (ARM TrustZone, Intel SGX) to attain stricter isolation guarantees. For example, ARM TrustZone eanbles virtual virtualization thanks to dual world execution model. However, these solutions are strictly linked to the specific platforms. Examples are LTZVisor/RTZVisor [21, 22] and VOSYSMonitor [23, 24].

4. **Unikernel**: runs on top of an hypervisor to enable the execution of a single application in its virtual domain, providing isolation, performance, and security. Examples are ClickOS [25] and HermitCore [26].

The selection factors for a hypervisor with industrial standards are its footprint, the compliance with industry safety-related standard, the software license, the explicit support to high availability, fault tolerance and security, and the supported HW platform [10]. Cinque et. al provides an extensive list of these solutions applied to the MCS in order to meet the industrial standards [10].

2.1.2.1 Bao

Bao (from Mandarin Chinese “bǎohù”, meaning “to protect”) is a security and safety-oriented, lightweight bare-metal hypervisor, developed by the ESRGv3 team at University of Minho, targeting the embedded real-time domain and especially the MCSs for Armv8 and RISC-V platforms. It follows the pioneer static partioning architecture of *Jailhouse* [13], and improves it by discarding the need of the Linux Kernel to boot and manage its VMs. From a security and safety perspective, this dependency compromises the

system by bloating the system Trusted Computing Base (TCB) and intercepting the chain of trust in secure boot mechanisms [14]. The MCSs certification process is also hindered by the size of and monolithic architecture of such OSs.

Jailhouse's breakthrough consists in a minimal software layer that leverages HW-assisted virtualization technology to statically partition all platform resources and assign each one exclusively to a single VM instance at instantiation time. The scheduler can then be discarded as each virtual core is statically associated to a single physical CPU, and the complex semantic services are relegated for the VMs, further decreasing the size and complexity. It provides strong isolation and real-time guarantees but at the expense of efficient resource usage requirement. However, the Linux dependency is a liability for safety/security and performance too.

Furthermore, the static partioning approach is not enough *per se*, due to HW resources sharing across partitions such as Last-Level Caches (LLCs), interconnects, and memory controllers, which breach temporal isolation, hurting performance and determinism [17, 27]. A malicious VM can exploit this by increasing their usage of a share resource (Denial of Service (DoS) attack) or by indirectly accessing other glsvm's data thorugh the implicit timing side-channels [28]. To tackle this issue, some techniques were implemented at both the OS and hypervisor level such as cache partioning (via locking or coloring), and memory bandwidth reservations [14].

Taking this into account Bao was implemented as clean-slate hypervisor (see Fig. 3), comprising only a minimal thin layer of privileged SW leveraging Instruction Set Architecture (ISA) virtualization support to implement static partioning of HW resources. Its most relevant features are: memory is statically assigned using 2-stage translation; I/O is pass-through only; 1:1 mapping of virtual to physical CPUs (no scheduler required); no external dependencies (except for standard platform management SW); provides a basic mecanism for inter-VM communication; Trusted Execution Environment (TEE) support for increased security [14, 29].

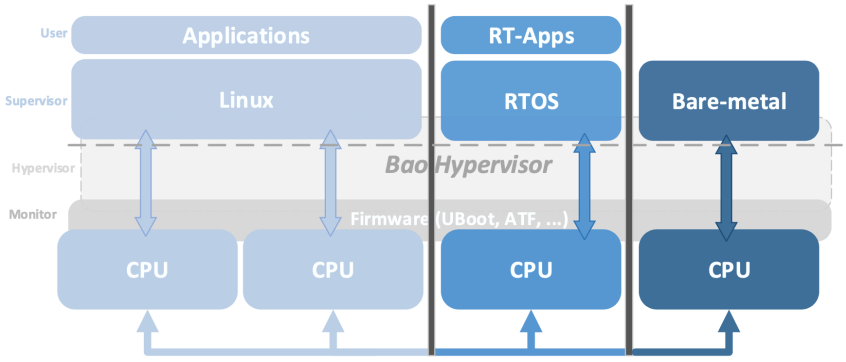


Figure 3: Hypervisor Bao architecture [14][3]

Guest isolation is critical for a MCS. Bao achieves complete logical temporal isolation due to the exclusive CPU mapping, discarding the scheduler, and the availability of per-CPU architectural timers

[3]Used under the terms of the Creative Commons BY 3.0 license.

directly managed by the guests. Spatial isolation is provided by the 2-stage HW virtualization support for the logical address space isolation. The translation overhead, page table and Translation Lookaside Buffer (TLB) pressure is minimized using superpages, whenever possible, faciliting speculative fetch for potential guest performance improvement. A page coloring mechanism was implemented to enable LLC cache partioning independently for each VM, but at the expenses of memory waste and fragmentation, and increased boot time [14].

The I/O are directly assigned to guests in a pass-through only configuration. For the memory-mapped IO architectures, as the ones supported, it uses the existing memory mechanism and 2-stage translation provided by the virtualization support to implement this for free. A peripheric can be shared among guests, as exclusive assignment is not checked [14].

The interrupt virtualization support is restricted to the Arm Generic Interrupt Controller (GIC)v2 and Arm GICv3, which does not support direct interrupt delivery to guest partitions. Instead, all interrupts are dispatched to the hypervisor, which then must re-inject them into the guest VM using a limited set of pending registers, leading to unavoidable increase in both interrupt latency and the complexity of interrupt management code [14]. This was fixed in the newest version of the specification GICv4 which bypasses the hypervisor for guest interrupt delivery [30].

It is also noteworthy to mention that Xen has recently introduced *Dom0-less*, eliminating the Linux dependency to boot and execute its hypervisor and VMs. Nonetheless, Bao currently provides the same features but with a smaller TCB and with clean security features. Moreover, and although being in its infancy, preliminary evaluation demonstrate only minimal virtualization overhead [14]. Lastly, Bao is open-source [31] due to the developing team's strong belief that security requires transparency. For all the aforementioned reasons, Bao is the hypervisor of choice to use in this work. Tab. 2 provides a summary of the Bao hypervisor.

2.2 Unmanned Aerial Vehicles

Unmanned Aerial Vehicle (UAV)s, Unmanned Aerial System (UAS), or more commonly drones, are a class of unmanned robotic vehicles that can execute flying missions and carry payloads, guided either by remote control stations or in an autonomous way [2, 3]. They belong to a broader class of Unmanned Vehicles (UVs), alongside with Unmanned Ground Vehicles (UGVs), Unmanned Surface Vehicles (USVs) (e.g., boats) and Unmanned Under water Vehicles (UUVs) [3].

UAVs date back to as far as the 18^{th} century. In 1783, France, the first uncrewed aircraft — a hot air balloon — was publicly displayed [2, 32]. In 1896, Alfred Nobel created the first camera-based UAV (rocket) and launched it. In 1935, U.K., the first modern UAV was developed, a low cost radio controlled aircraft. The term *drone* was presumably coined by Lieutenant Commander Fahrney, who as in charge of U.S. Navy program *Radio Controlled Aircraft* [32]. The first UAV designed for surveillance and scouting was developed in 1973 in Israel, and the *Gulf War* was the first conflict utilizing UAVs. Starting from 2003,

Table 2: Bao hypervisor summary

Architecture	**Supported platforms**	**Size**	**License**	**Version**
Type I, Static	Armv8 RISC-V (experimental)	Small ~5 kLOC C + ~500 LOC ASM	GPLv2	0.1.1.
Features	**Isolation**			
Static partioning IO pass-thorugh only 1:1 mapping of virtual to physical CPUs (no scheduler required) Virtual interrupts directly mapped to physical ones No external dependencies (except for standard platform management firmware) Provides a basic mecanism for inter-VM communication Trusted Execution Environment support for increased security State-of the art partioning mechanisms to be implemented (e.g., memory throttling)	**Spatial** Provided by a 2-stage translation HW virtualization support Translation overhead is minimized using superpages whenever possible, faciliting speculative fetch for potential guest performance improvement It uses cache coloring to enable LLC cache partioning independently for each VM, but at the expenses of memory waste and fragmentation and increased boot time		**Temporal** Exclusive CPU assignment discards the scheduler Per-CPU architecture timers are available to and directly managed by the guests	
IO	**Interrupts**		**Related Work**	
Directly assigned to guest in a pass-through only IO configuration For memory-mapped IO it uses the existing memory mechanism and 2-stage translation provided by the virtualization support A peripheric can be shared among guests (exclusive assignment is not checked)	Currently supports only Arm GICv2 Interrupts are forwarded to the hypervisor which must re-inject them in the VM using a limited set of pending registers Fixed in GICv4 which bypasses the hypervisor for guest interrupt delivery		Jailhouse: pioneer in the static partioning adopted by Bao but requires the Linux Kernel to boot Xen Dom0-less: eliminates the Linux dependency to boot and execute Bao provides the same features from the previous ones, but has a smaller TCB and implements clean security features	

the UAV commercial market started to emerge with the release of the *Amazon Prime* [2]. However, it was until 2006 that the UAVs were first permitted in the U.S. civilian space. More recently, in 2010, the first smartphone controlled quadcopter was developed, and in 2013 camera equipped UAVs entered the consumer market [33].

From then on, the selling price dropped significantly, alongside with the emergence of some of the first open-source projects on UAV control — ArduPilot in 2008 [34] and Dronecode [35] (now PX4) in 2011 — led to a boom in the commercial UAV market. In 2017 the North America market had a revenue of 737 million USD dollars and a eight-fold increase is expected for 2026 with a staggering 6.7 billion USD dollars, with strong contributions from the sectors of agriculture and farming, and security and law enforcement [4].

The versatility and utility of UAVs are well displayed is by its wide range of applications, performing tasks with high added value and that would be somewhat hard or impossible for a person to achieve: rescue operations and saving lifes, agriculture and farming, building structures, pipeline inspections, delivering goods and medical supplies, video capturing and filming, surveying, inventory management, providing telecommunications in remote areas, among others [2].

However, only recently regulations have been explicitly enforced on UAVs, with many countries allowing drones to fly over populated areas (at altitudes lower than 150 m) [5]. For example, in 2019, the E.U. stated

that all drones under 25 kgs are able to fly without prior authorization under some constraints. On the other hand, it imposed that all drones must register in their respective states, and that each state must define no-fly zones where drones are forbidden to enter [36]. Broadly, five important categories must be considered when analyzing the UAV regulations [37, 38]:

1. **Applicability**: refers to the applicable scope of UAV regulations, typically including type, weight, and role of the UAV;
2. **Operational limitations**: restricts the locations for UAV operations.
3. **Administrative and legal requirements**: set of rules and regulations to monitor the use of UAVs.
4. **Technology specifications/requirements**: mechanical, communications and control capabilities that ensure its safe operation.
5. **Moral and ethics**: refers to the privacy and security of people in general.

Furthermore, no global regulation for UAV standardization exists. Thus, in 2020 Federal Aviation Administration (FAA) in collaboration with National Aeronautics and Space Administration (NASA) and other agencies published the UAS Traffic Management (UTM) 2.0, describing protocols for enabling multiple, Beyond Visual Line-Of-Sight (BVLOS) drone operations with the same airspace.

2.2.1 Classification

An UAV can be classified in multiple ways:

- **Thrust forces & Flight principles**: UAVs can be lighter than air, e.g., a balloon or a blimp if it uses a motorized propulsion system. On the other hand, if they are heavier than air, they are named gliders, rotor-crafts, and birds [4].
- **Airframe**: probably the most noticeable external feature of a UAV is its chassis, or airframe. Several airframes exist for different applications from three main types: fixed-wing, rotor, and hybrid. Fixed-wing can travel faster than all other types of UAVs due to its aerodynamics and propulsion system, can carry heavy payloads, and have long autonomy. However, they require takeoff and landing from a runway. They are typically used for power line inspections and aerial mapping. Single rotor and multirotor UAV can hover and do Vertical Take-Off and Landing (VTOL). The single rotor long autonomy, but are expensive, require skilled operators, and are mechanically complex and vulnerable to vibrations. Multirotors, on the other hand, are the cheapest and most manufactured ones, but they typically have low autonomy since they consume more power. They can have a varied number of propelers, such as tricopter, quadcopter, hexacopter, and octocopter [4].

- **Altitude**: Broadly, UAVs are divided into Low Altitude Platforms (LAPs) and High Altitude Platforms (HAPs) . A LAP maximum altitude ranges from 3 to 9 kms and is typically used to support cellular communications. HAPs, on the other hand, are deployed above the 9 km altitude and are used to support cellular communication but with wider coverage, endorsed by companies such as Google and Facebook [4].

- **Overall weight**: UAVs can have few grams of weight or hundreds of kilograms. For example, Australia labels them as *micro* (below 100 g), *very small* (from 100 g to 2 kg), **small** (from 2 to 25 kgs), **medium** (from 25 to 150 kgs), and **large** (over 150 kgs) [2].

- **Power source**: UAVs can be powered using electricity, fuel, or a hybrid solution. Except for the fuel-only powered ones (e.g., Nitro Stingray, and Goliath Quadcopter [39]), all the others use electric motors. These motors can be powered through batteries (e.g., Parrot[40], Dji Mavic 3 [41]), typically Lithium Polymer (LiPo), Hydrogen Fuel Cells (HFCs) (e.g., Energyor H2Quad 1000 [42]), and solar power. The hybrid solutions use fuel + batteries (e.g., Flaperon MX8 [43]) or HFC + batteries, and are a compromise between the two power sources. Batteries typically have the shortest autonomy and are heavy and bulky, while fuel, although not a clean power source, has the highest power density, leading to higher autonomy. The hydrogen fuel cells are a intermediate solution, but are typically more expensive than batteries and have more complex power management.

- **Target audience**: UAVs are developed with a specific target audience in mind, namely the recreational/hobbyist, the commercial, and military one.

- **User-modifiable**: In 2021, 26% of all commercial drones sold were open-source (e.g., Parrot) [44]. There is an increased trend for open-source adoption due to higher transparency, extensability and usage from the end-user perspective, and due to bootstrapping opportunity since the developers can leverage from the ecosystem, e.g., using the flight control algorithms which are hard and costly to develop. At the other end of the spectrum lie proprietary drones (e.g., Dji), which hold the highest market share. Transparency is compromised and extensibility is typically provided using a Software Development Kit (SDK) [45].

Fig. 4 illustrates the several types of UAVs, focusing on airframe and power source. The most prominent UAV's characteristics are speed, autonomy, payload, range, and altitude. The speed depends mainly on the propulsion system, aerodynamics, weather conditions and power usage. Typically, a small UAV can reach speeds up to 50 km/h, while a large one can reach speeds up to 360 km/h[4].

The autonomy or endurance, refers to the maximum flight time with a single charge (battery, fuel, or both), varying from a 20–30 minutes (small UAVs) to several hourse (large UAVs). Size, weight and weather conditions have a strong impact on the autonomy. Some technologies are currently being develop to support in-flight recharging through renewalable sources (Photovoltaic (PV) cells) or wireless power techniques (e.g., laser emission)[4]. The payload refers to the lifting capability of the UAV for load carrying,

(a) Firebird: Fixed-wing [46]

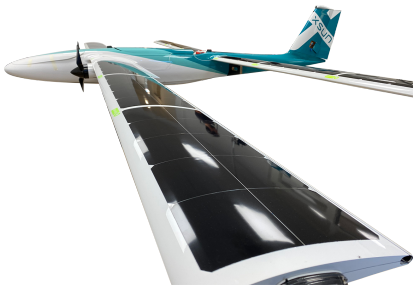
(b) Velos: single-rotor [47]

(c) Parrot: multirotor [40]

(d) DeltaQuad Pro: fixed-wing hybrid [48]

(e) Energyor Quad 1000: hybrid (HFC + batteries) [42]

(f) HAMR: hybrid (fuel+batteries) [49]



(g) Solar X One: solar powered [50]

Figure 4: UAV types

varying from few grams (small UAVs) to hundreds of kilograms (large UAVs). Most common payloads are sensors and video cameras [4]. The range defines the distance from where the UAV can be controlled remotely and depends on the communication technology and network, and the weather conditions[4]. The altitude is the height a drone can fly, weather by technological constraints or by legal ones [4]. Fig. 76 depicts the UAV's generic overview, summarizing its main concepts.

2.2.2 System overview

Fig. 5 shows an overview of the UAV system and its ecosystem. Positioned at the center of the figure (1) are the UAV and its associated flight control HW and SW. Its main tasks are path planning, communication management, data acquisition, and mission [51].

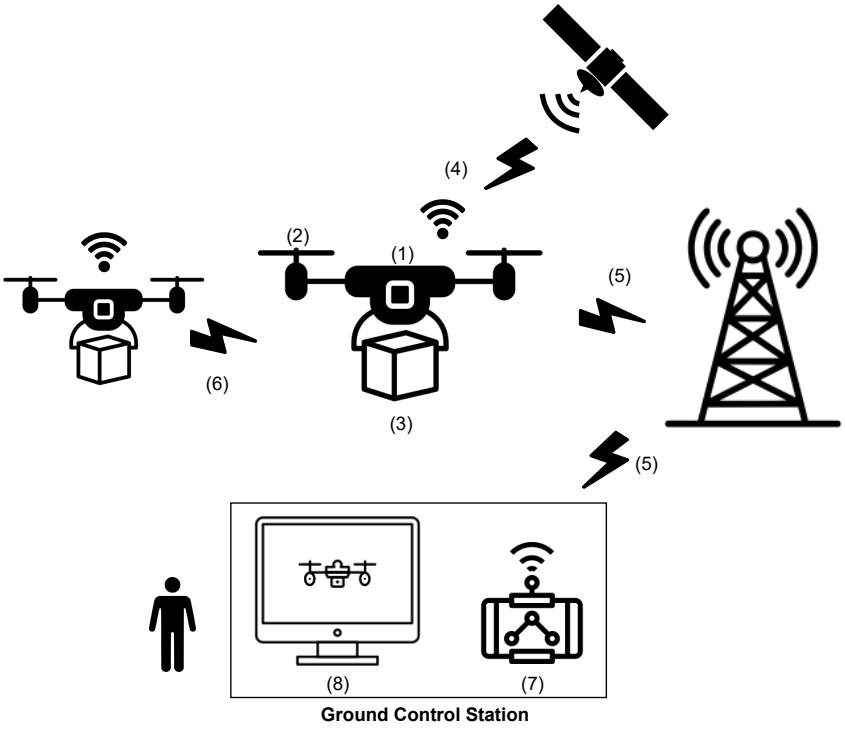


Figure 5: UAV system overview

The path planning works in combination with the Ground Control Station (GCS) to assist in the navigation, finding the optimal path, while providing environmental awareness through weather and climate monitoring, and controlling the motion and speed for obstacle avoidance. To achieve this, the on-board flight controller collects data from multiple sensors (e.g., ultrasonic/infrared sensors for obstacle avoidance, Inertial Measuring Unit (IMU), barometer, Global Navigation Satellite System (GNSS)/Global Positioning System (GPS) module, etc.) and, together with the commands received from the GCS, implements attitude estimation and the control law (e.g., Kalman filter) to drive the propulsion system (motors) (2).

The flight controller also manages the communications between three types of links [51]: UAV to GCS (5) – radio communication used for transmitting instrument readings such as audio or video and for human remote control (7) (8); UAV to Satellite (4) – carries weather, climate, and GPS information required for accurate UAV navigation; UAV to UAV (6) – can be used for cooperative missions or to provide environmental awareness (e.g., signal danger). The mission refers to the flight's goal, e.g., video capturing, topographic mapping, etc., and is directly related to the payload (3) (e.g., camera, Light Detection And Ranging (LiDAR), etc.).

Sensors can be typically categorized into obstacle avoidance, payload, and navigation [52]. Ultrasonic

and infrared sensors are generally used to avoid collisions with obstacle, but Time-of-Flight (TOF) sensors like LiDAR can also be used, although more expensive and more complex. The navigational sensors include the IMU — used to estimate orientation and heading of the vehicle, the barometer — used to estimate of the UAV with a precision of few centimeters, and the GNSS/GPS module — used to estimate the drone's geolocation for navigation and autonomous flight with a precision of up to 5 meters, thus requiring the IMU and barometer to improve accuracy [53].

The actuators depend on the propulsion system, although electronic drives are always used. For a typical multi-rotor UAV, Brushless Direct Currents (BLDCs) motors are used for rotors, with an Electronic Speed Controller (ESC) — an glsac/Direct Current (DC) power converter and high-frequency variable motor speed controller. Fixed-wing hybrids include also servomotors for flaps [54]. Fig. 77 depicts the concept map of UAV's tasks and components.

UAVs can be organized into a top-down hierarchy of layers [3], as follows (see Fig. 78):

1. **Flight Supervision**: the topmost layer handles flight's management, safety/authorization, and remote UAV's identification. Flight management is achieved through the transmission of commands from the GCS to the UAV's Flight Control System (FCS). The GCS is software running on ground-control devices, and they typically provide path planning, mapping and real-time flight statistics superimposed on a map. The most notable example is probably `QGroundControl`, which provides full flight control and setup for UAV vehicles, cross-platform, and with a mobile-styled User Interface (UI). It supports actions to be triggered in case of failures (fail-safe features), such as geofencing (limits the permissible fly areas). Safety/authorization handles the authorization to fly (if required) and its safety. Although autopilots, like PX4 and ArduPilot, and the GCS exist, providing several safety features, the pilot in control is ultimately responsible for ensuring the safe flight of any UAV. As such, it must enforce the use of failsafe actions, like in the case of low battery, loss of Radio Controller (RC) signal, data link failures, or geofence violations, and request authorization to fly if legally demanded. For example, pilots can use `LAANC` Application Programming Interfaces (APIs) or `Google Wing OpenSky` to receive FAA authorization for entering controlled airspace. On the other hand these softwares provides the air traffic professionals drones operations' visibility. Remote identification will be required for all UAVs beginning in 2023. Thus, all UAVs must be equipped to a Remote IDentifier (RID), an unique electronic identifier comparable to a vehicle license plate, with direct broadcasting of the RID to anyone within range of signal.

2. **Command & Control**: the second layer ensures drones can fly safely, through the transmission of commands to the UAV, yielded by an human operator of computer-generated via autopilot. The command is sent to the UAV via proprietary protocols or open APIs (e.g., MAVLink SDK or Parrot SDK). The autopilot consists in SW stack running on the FCS. Noticeable open-source autopilots are `PX4`, `ArduPilot`, and `Paparazzi`.

3. **System Simulation**: UAV behavior is analyzed in respect to different environment and conditions,

mainly through flight modeling, traffic modelling — models real-world traffic interactions (land and air) — and network communication modelling.

4. **Operating Systems**: OSs are used to interface the electronic HW that controls UAV operation through a tractable abstraction. UAVs can a myriad of OSs, both open-source — such as Robotic Operating System (ROS), `Linux`, `FreeRtos`, `NuttX`, `ChibiOS` — or proprietary, like `VxWorks`.

5. **Physical HW**: comprises the actual electronic HW of the UAV, such as sensors, motors, communications, etc.

2.2.3 Security and Safety

Another very important feature, often overlooked, is the UAV's ecosystem security, and alongside with it safety of people and goods [55]. Security is often not embedded in the system's design, posing all sorts of problems. For example, UAVs often include onboard wireless communication modules that use open, unencrypted, and unauthenticated channels, exposing them to a variety of cyber-attacks [56, 57]. The problem is not restricted to commercial applications, with the U.S. army banning Dji drones — the most widely used ones by the army — for cybersecurity concerns in 2017 [58]. Hacking of drones is another major concern, exposing sensitive information and control of the UAV. In fact, several incidents have been reported in the media where drones were weaponized [59–61], and the volume and risk of such incidents are likely to increase significantly with the expected drone market growth in the foreseable future [4] and the new regulations adopted by many countries which allow drones to fly over populated areas [38].

The most common attacks to the UAV are DoS and Distributed Denial Of Service (DDoS), causing resource availability challenges which can be exploited to drain the batteries, overload the processing units, and flood the communications link, leading to huge services' interruption [4]. But more sophisticated attacks are used too, like GPS spoofing — impersonating as a valid GPS satellite to provide false data, GPS jamming, instrument spoofing (gyroscope, compass), or even killing the main process [5]. Ground control stations are a target too, since the attacker can indirectly obtain access to the UAV, usually through key loggers, viruses, and malwares, and steal all of its data or send malicious and erroneous commands to the UAV [4].

Nassi et al. [5] conducted a thorough and very extensive survey on the security and privacy issues associated with UAVs, the attacks and countermeasures. The authors divided the security attacks on two categories:

- **Targetting UAVs**: corresponds to an attack an ill-intentioned civilian performs on an UAV using, e.g., an Software-Defined Radio (SDR), a computer or a commercial laser. Six targets were identified — the UAV electronic HW, the UAV chassis and package (e.g., propelers and cargo), the GCS, the First-Person View (FPV) channel, the pilot and cloud services/servers. Attacks on the UAV HW include installation of fake firmware, instrument spoofing and GPS jamming. Countermeasures vary with each attack, but, for example, the installation of fake firmware on the device can be

opposed by requiring the digital signature of the firmware before installing. Direct attacks to interrupt UAV's flight control and commiunications links to modify mission parameters can be mitigated through onboard SW and HW mechanisms, such as real-time monitoring, instantaneous estimation of the controller, alert warning and immediate action on any alteration from the intended controller model [4]. Attacks on the chassis and package include fake Computer-Aided Design (CAD) files to undermine its fabrication and deployment, especially critical for open-source HW, nets, and bullets. They can be opposed by requiring digital signature and by using parachutes, respectively. The FPV channel is the radio communication channel between the UAV and the GCS, which enables the pilot to fly as if it was on board and consists of a downlink — used for video streaming — and an uplink — used to control the UAV via the GCS. The most known methods are pilot's deauthentication, taking and dowloading videos and pictures, killing the main process, or remote control of the drone, which can be mitigated through the usage of an encrypted network protocol. Jamming is also an issue and can be remediated using channel hopping. Lastly, cloud services and servers expose the UAV indirectly to hackers, because, although most UAVs are not connected to the Internet, the GCS is, sending the telemetry of flights to cloud servers for storage and analysis. The methods used to explore this link are deanonymizing pilots and extracting flight history, which can be both mitigated through the implementation of authorization and two-factor authentication mechanisms.

- **Targetting people**: UAVs can attack people, on a individual, organization or nation base. The countermeasures include detection and tracking, assessment, and interdiction. Several detection methods exist, such as radars, video and infrared cameras, LiDAR, acoustics, acoustics and optics, etc. However, it is important to note that none of these methods are fail-proof. For example, specific radars have to be used with very specialized operators, since common radars are designed for large aircraft detection and drones uses materials which are not very radio reflective. Another example is the acoustics, where sensors exist to detect the specific spectrum of the drone's propellers, but can be easily circumvented by going into silent (stealth) mode [62]. Assessment concerns the rating and identification of hostile drones, especially important in areas where drones are used for both legal and illegal activies. The most used methods are based on UAV's classication, to identify the manufacturer and model of the drone. In the event of the detection and assessment of a hostile drone, interdiction must be applied, i.e., the drone should be disabled. For this purpose, several methods can be used like bullets and nets, which are dangerous and not very effective, commercial jammers, predator birds and laser cannons.

Safety failures are also very serious, since they pose risks on the integrity of people and goods. They can be categorized as follows [63]:

- **Damague due to calculation errors**: occurs when a UAV is more dangerous than planned. For example, in 2020 in the U.K., an activist group piloted hundreds of drones with the 3-mile depletion zone to interrupt the flights, which could have been catastrophic.

- **Sensor failures**: UAV sensors do not easily detect delicate objects, such as wires, tree branches, and transparent surface such as buildings windows. Falls by collisions with these types of objects are very common due to undetected obstacle or pilot's excess of confidence.

- **Obstacle deviation**: collisions and crashes caused by object deviation are common, from building to trees.

- **Direct attacks**: these attacks are performed to harm people, for personal or ideological reasons, or to steal the UAV payload, e.g., an organ, food, or vaccines. The recreational/hobbyist UAVs are the most commonly targets.

- **Accidental damage**: can occur when a UAV gets out of control, due to legitimate loss of control by the owner, cyber attack, or SW or HW malfunction. Indepently of the cause, the extent of the damage on people as goods can be severe, due to crash of UAV falling from the sky.

Clearly, security and safety are fundamental aspects of the UAV's operation, but they require more effort to decrease the attack vectors and the attack surface. Ferrão proposed that UAV's design considers simultaneously both aspects, as they strongly influence each other [63]. Fig. 79 illustrates the UAV's security and safety taxonomy.

2.2.4 UAV Reference Hardware

In this section the reference hardware for UAVs is discussed. An overview and the HW architecture is presented. Then, the open-source and commercial solutions are discussed and compared.

2.2.4.1 Overview and Architecture

Fig. 6 illustrates the high-level abstraction of the UAV HW architecture [55, 64]. The main computing platforms are depicted in orange: the flight controller (FCS), e.g., Pixhawk, and, optionally, the companion computer, which provides extra functionalities like navigation — avoidance and collision prevention, mission-related features — associated to the payload, e.g., camera surveillance, topographic mapping, etc. — or telemetry data processing, off-loading the FCS.

The FCS is typically composed of a main processor, for UAV control, and a optionally fall-safe coprocessor, dedicated to fail-safe features, i.e., the actions that must be triggered in case of a failure, e.g., returning to base when the battery reaches the low threshold. The main processor handles the communications — commands arriving from the manual RC controller or between the companion computer, the control of the aircraft — typically using Proportional Integrative Derivative (PID) models and/or estimation methods (e.g., Kalman filter), and the I/O — to interface the critical sensors, such as IMU, barometer, gyroscope, and compass, or the actuators, such as BLDC motors and servomotors. The power system supplies the required energy for computing and I/O.

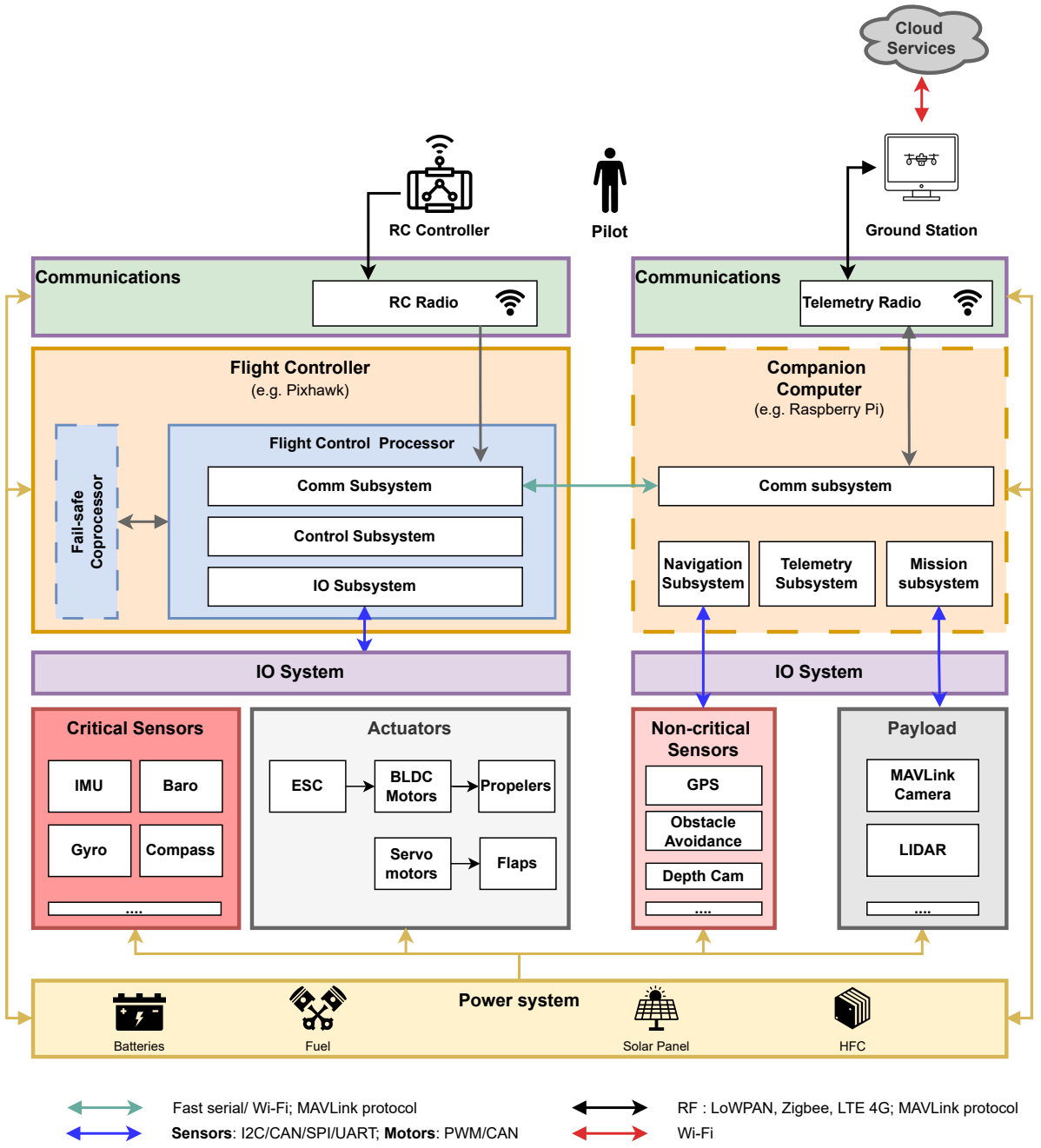


Figure 6: UAV HW architecture: high-level abstraction

As aforementioned, the flight controller does not explicitly require the companion computer, since the commands sent manually by the pilot through the RC controller are sufficient to control the aircraft flight (Line-Of-Sight (LOS) navigation). However, for autonomous missions — autopilot — the non-critical sensors like the GPS, and obstacle avoidance sensors, are required. In this mode, the companion computer sends telemetry data to the GCS, which are typically superimposed on a map for straightforward navigation and receives the navigational and mission-related commands, processing them and dispatching the lower level commands to the flight controller or I/O system.

2.2.4.2 Open-source solutions

Open-Source Hardware (OSH) solutions have been developed to provide more transparency, extensibility, flexibility, maintenability, while trying to be cost-effective. Open-Source Hardware (OSH) means that the HW design (i.e., mechanical drawings, schematics, Bill Of Materials (BOM),Printed Circuit Board (PCB) layout data, Hardware Description Language (HDL) source code), and the SW that drives the HW, are all released under free/libre terms [65]. The user can purchase or manufacture the HW components individually or in Commercial Off-The-Shelf (COTS) kits.

The OSH solutions fall into three main categories: ARM-based platforms, Atmel-based platforms, and RaspberryPi-based platforms [53]. However, the currently active OSH projects are based only in ARM platforms, such as, `Pixhawk 4`, `Paparazzi Chimera`, `CC3D`, and `CUAV v5 Plus`, among others.

Pixhawk is a company that develops open standards for drone hardware, providing readily available HW specifications and guidelines for drone development. `Pixhawk 4` is an advanced autopilot designed in collaboration with Holybro HW manufacturer and the PX4 (open-source autopilot SW) teams [66], released under the Berkeley Software Distribution (BSD) license (see Fig. 7). It is optimized to run PX4 v1.7 and later and is suitable for academic and commercial developers. It runs PX4 on top of the NuttX OS [66]. It consists of two 32-bit ARM processors for the FCS (Flight Management Unit (FMU), 32-bit Arm Cortex M7 @ 216 MHz, 2 MB memory, 512 KB Random Access Memory (RAM)) and I/O (32 bit Arm Cortex-M3 @ 24 MHz, 8 KB Static Random Access Memory (SRAM)), respectively [66]. It comes with on-board sensors (accelerometers/gyroscopes, magnetometer, and barometer) and a GPS module. It has Pulse-Width Modulation (PWM), Controller Area Network (CAN) Inter-Integrated Circuit (I2C), Universal Asynchronous Receiver-Transmitter (UART), Serial Peripheral Interface (SPI), and telemetry interfaces (radio and companion computer) [66].

The `Paparazzi Chimera` is an OSH flight controller released under the GNU Public License (GPL) license [67] (see Fig. 8). This flight controller integrates all I/O in the same board, and it uses the ARM Cortex-M7 STM32F767 @ 216 MHz (2 MB Flash, 512 Kb SRAM) Micro Controller Unit (MCU). It comes with on-board sensors (IMU, barometer, and pressure sensure), an XBEE modem holder for communications, and a dedicated serial link and power supply for the companion computer (e.g., Beaglone, RaspberryPi, etc.). It has PWM, CAN I2C, UART, SPI, and servomotors interfaces [66].

The `CC3D` is an OSH flight controller released under the GPL license [68] (see Fig. 9), running the `OpenPilot` firmware. It uses a STM 32-bit MCU @ 90 MHz (128 kB Flash and 20 kB SRAM). It comes with on-board gyroscope and accelerometers, support to serial and Universal Serial Bus (USB) telemetry, up to ten motors, and provides camera stabilization.

`CUAV v5 Plus` is an advanced autopilot supporting the `ArduPilot` firmware [69], released under the BSD license (see Fig. 7). It consists of a 32-bit ARM processor for the FCS (FMU, 32-bit Arm Cortex M7 @ 216 MHz, 2 MB Flash, 512 KB RAM) and a 32-bit IOMCU coprocessor. It comes with on-board sensors (accelerometers/gyroscopes, magnetometer, and barometer) and a GPS module. It has PWM, CAN I2C, UART, SPI interfaces [69].

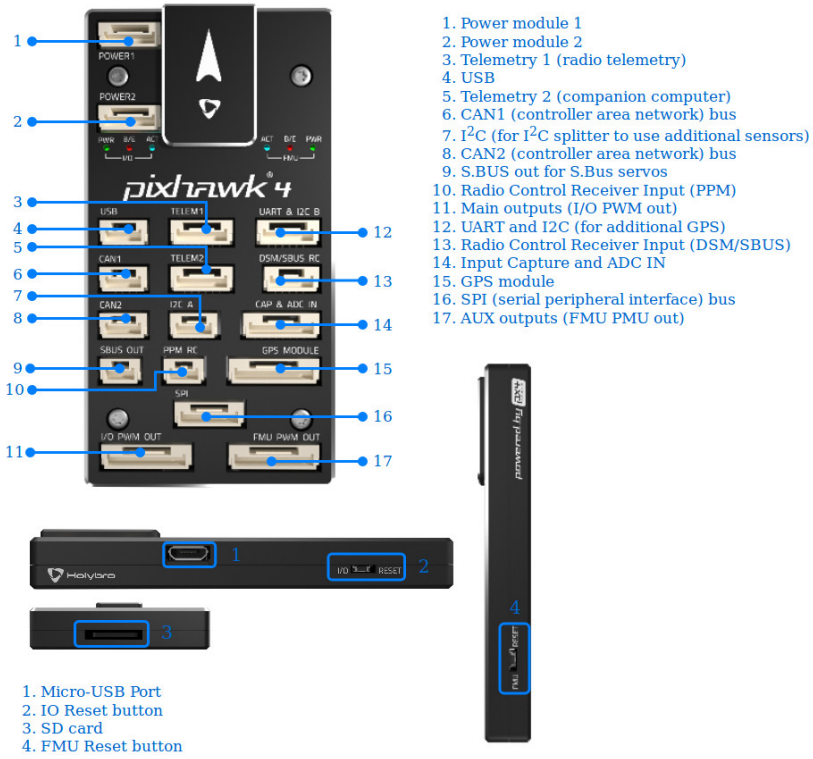


Figure 7: Pixhawk 4 flight controller [66]

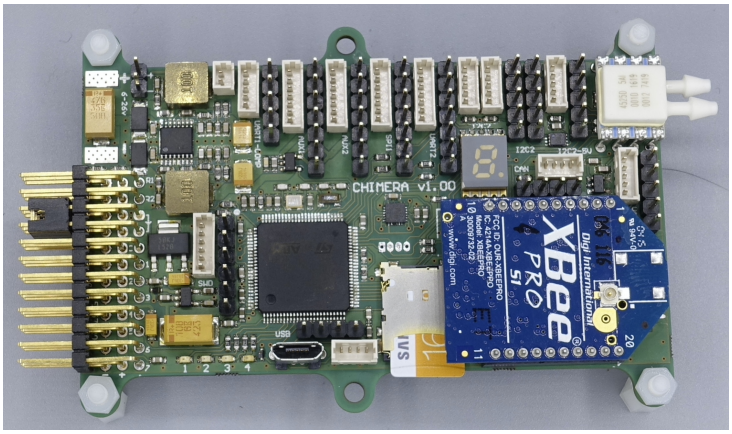
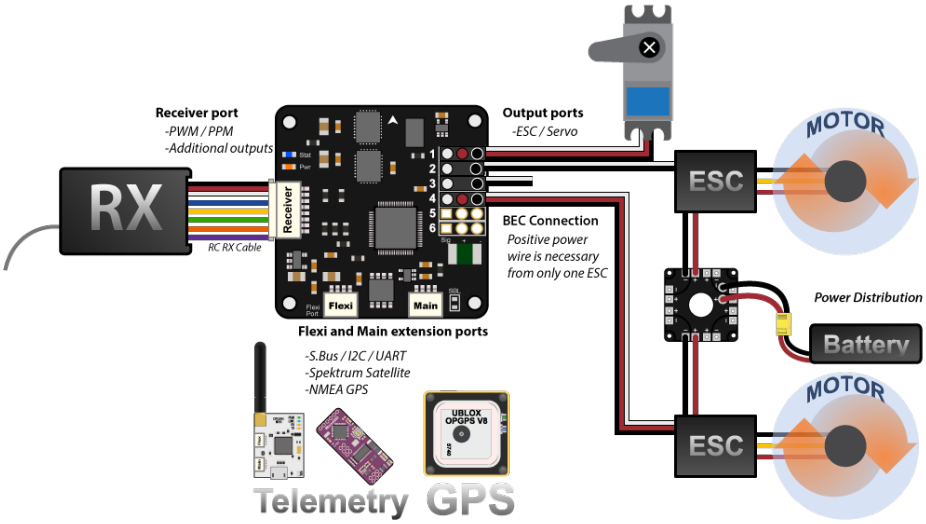


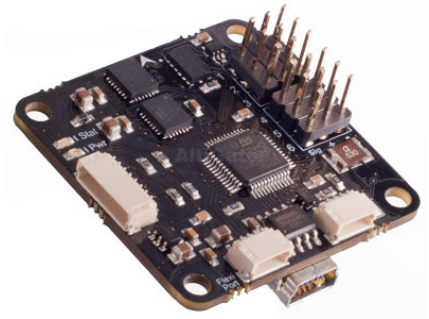
Figure 8: Paparazzi Chimera flight controller [67]

2.2.4.3 Commercial solutions

The commercial term here refers to the proprietary nature of the HW, opaque to the end-user. The lack of transparency can be an issue, raising suspicions, like the on-going ban of the U.S. Army to the Dji drones [58, 70]. Nonetheless, proprietary solutions represent 70% of the market share, with Dji being the biggest manufacturer [44]. The commercial solutions can be classified into three main categories: microcontroller-based platforms, Field-Programmable Gate Array (FPGA)-based platforms, and Companion Computer-based platforms.



(a) Connection diagram



(b) flight controller board

Figure 9: CC3D flight controller [68]

Figure 10: CUAV v5 Plus flight controller [69]

Microcontroller-based platforms The `SPRacing H7 Extreme` is a low-end flight controller running the `ArduPilot` SW stack (see Fig. 11). It features an STM32F750 ARM 32-bit MCU @ 400 MHz, and comes with on-board sensors (dual IMU, barometer). It supports I2C, UART, SPI, micro USB, and camera interfaces [71].

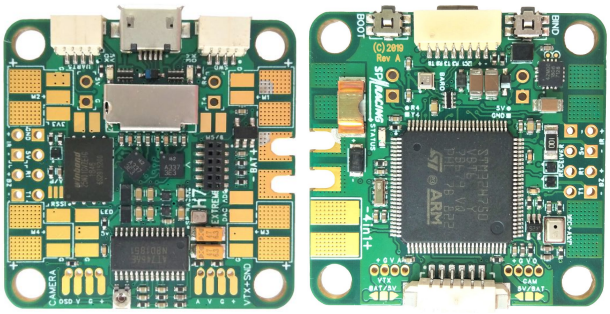


Figure 11: SPRacing H7 extreme flight controller [71]

FPGA-based platforms The `Aerotenna OcPoC-Zynq Mini` is a FPGA + ARM System on Chip (SoC) based flight control platform (Xilinx Zynq Z-7010), which supports ArduPilot and PX4 SW stacks [72] (see Fig. 12). The FPGA I/O's flexibility allows for rapid sensor integration and customization of the flight controller HW, adding capabilities such as triple redundancy in GPS, magnetometers, and IMUs. The FPGA is an Artix-7 with 28k logic cells, and the ARM SoC is ARM A9 dual-core @ 667 MHz, including 512 MB RAM and 128 MB of flash memory. It requires an Storage Disk (SD) card of 16 GB for Linux booting and data logging. Includes also on-board sensors (dual IMU, barometer), 16 programmable tri-pin I/O and 10 programmable I/Os supporting the I2C, USB, SPI, CAN, Camera Serial Interface (CSI), and Generic Serial Interface (GSI) interfaces.

Figure 12: Aerotenna OcPoc-Zynq Mini flight controller [72]

Companion Computer-based platforms The `Navio2` is a RaspberryPi-based autopilot, which supports ArduPilot and PX4 SW stacks [73] (see Fig. 13). Basically, it is an UAV extension board (shield) for the RaspberryPi 2 or later, a quad-core 1 GHz computer running a real-time Linux and ArduPilot flight stack. The shield includes on-board sensors (dual IMU, barometer), a GNSS receiver, an RC I/O coprocessor, extension ports exposing Analog to Digital Converter (ADC), I2C, and UART interfaces for sensors and radios, 14 PWM servomotors outputs, and a triple redundant power supply.

The `PixC4-Jetson` (Fig. 14) is a high-end FMU, powerful single board computer and peripheral support system (USB, MIPI, Ethernet, M.2 slot, etc.) in a small form factor, designed to be integrated into end-user platforms [74]. It supports the ArduPilot and PX4 SW stacks. The term "PixC4" is derived

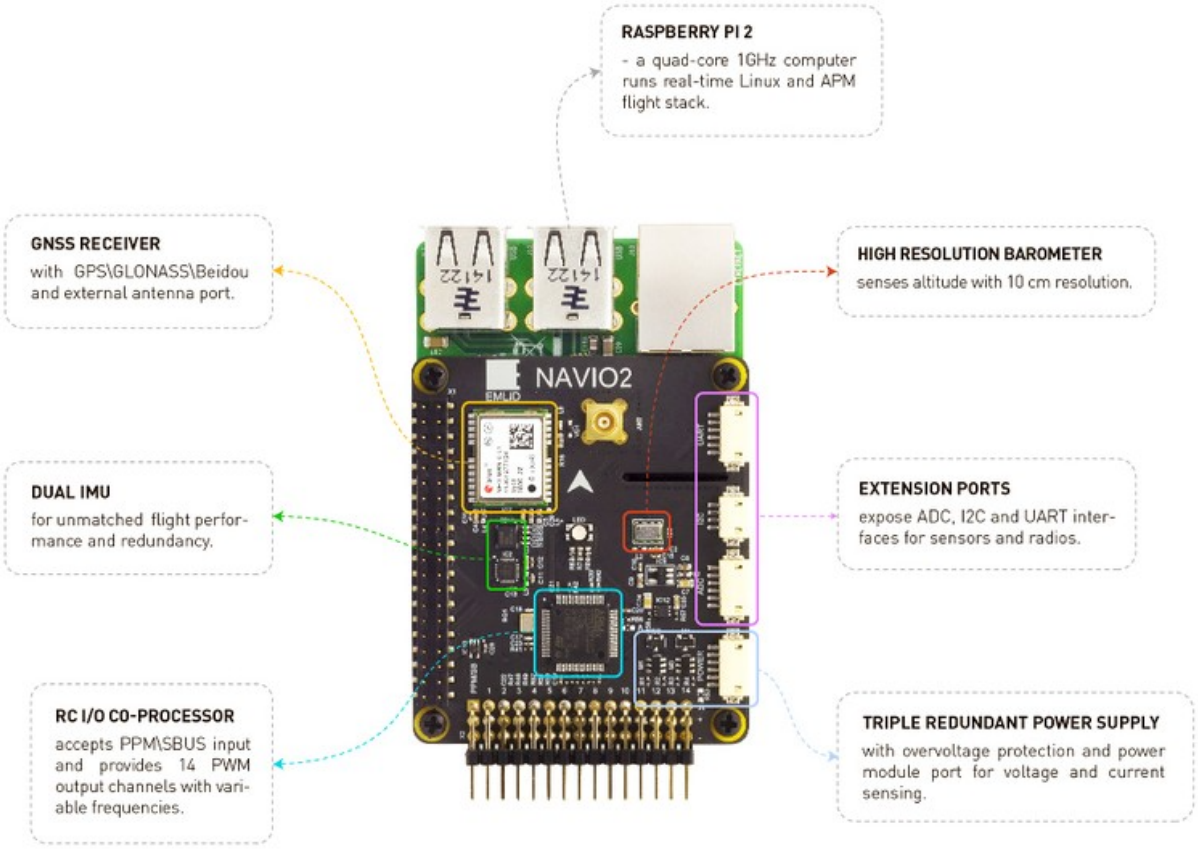
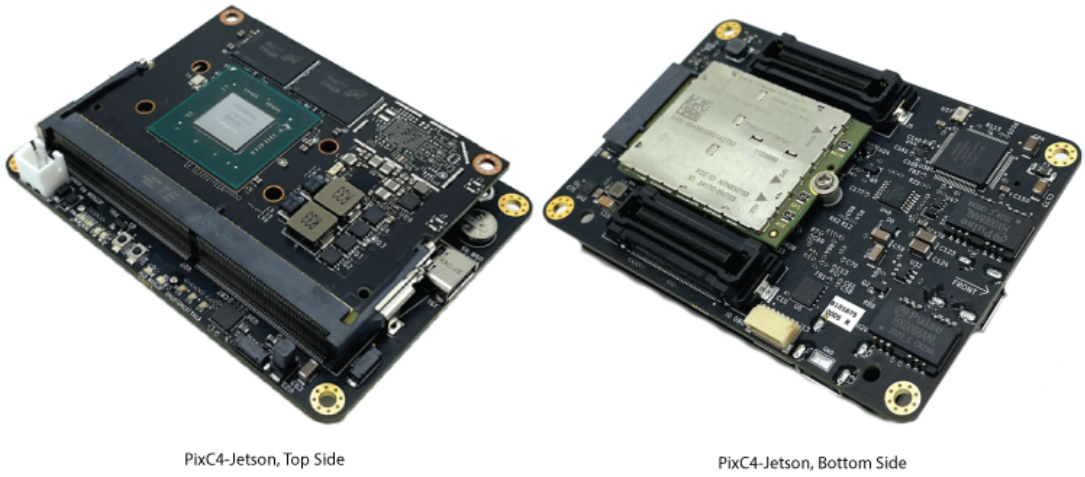


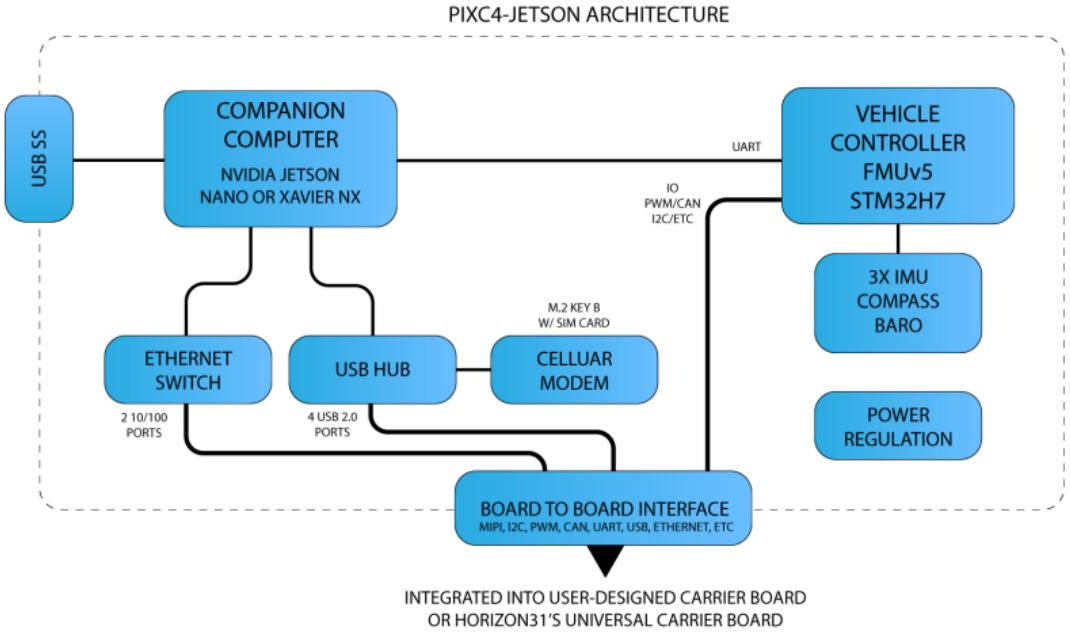
Figure 13: NAVIO2 flight controller [73]

from the Pixhawk, on which the FMU design is based (FMUv5) and C4, representing Command, Control, Compute and Communication. It is available as a turnkey solution including software pre-flashed on the Nvidia Jetson companion [74], enabling the following features: User Datagram Protocol (UDP) telemetry (MAVLink); Long-Term Evolution (LTE) connection management with Layer-2 peer to peer Virtual Private Network (VPN); multiple-endpoint video encoding pipelines, web interface for configuration and remote terminal access; scalable and secure cloud connectivity to Horizon31's U.S. servers and optional access to their cloud GCS and low-latency webReal-Time Communication (RTC) video distribution system. It feaures the STM32H743 processor with STM IO coprocessor, integrated Nvidia Jetson companion computer, ethernet switch (3 port), USB 2.0 Hub (7 port), cellular/LTE Modem support, and on-board sensors (IMU, barometer, and compass). It supports the PWM, I2C, UART, SPI, CAN, and CSI. It is important to note that the FMU derives from the Pixhawk FMUv5, but can be sold commercially as a proprietary solution because it is released under the BSD license, which does not obligate to make the modifications to the design open to the public. Obviously, this is very attractive for companies who want to commercialize drones, but do not want to invest the time and money to developed an end-to-end solution.

Another relevant option in the commercial market is the `Auterion Skynode X`, combining a flight controller, a mission computer and LTE connectivity all in on product [75] (see Fig. 15). Its compact form factor enhances ease of integration and optimises size, weight and power without compromising on



(a) Board



(b) HW architecture

Figure 14: Horizon31 PixC4-Jetson flight controller [74]

capability. The FMU (FMUv6X) is based on Pixhawk FMUv6 architecture (BSD license), comprising a flight controller – STM32H753 microcontroller (2 MB Flash, 1 MB RAM) – a I/O coprocessor – STM32F103, and a triple-redundant sensor architecture to minimize failure risk. It supports up to 8 UART, a dual redundant CAN, 100Base-TX Ethernet, 16 PWM outputs, 2 I2C and 1 SPI connections. The FMU runs an enterprise-hardened version of PX4, labelled as Auterion PX4 (APX4).

The mission computer comprises a ARM Cortex-A53 Quad core CPU running at 1.8 GHz and two embedded Graphics Processing Units (GPUs) – GCNanoUltra for 3D acceleration and GC320 for 2D acceleration – with 4 GB RAM and 16 GB (embedded Multi Media Card (eMMC)) + 128 GB (internal SD card) of storage. It supports Wi-Fi and Bluetooth 5 for low range wireless communications and a 4G LTE module with 50 MBit/s upload and 150 MBit/s download for long range communications. It

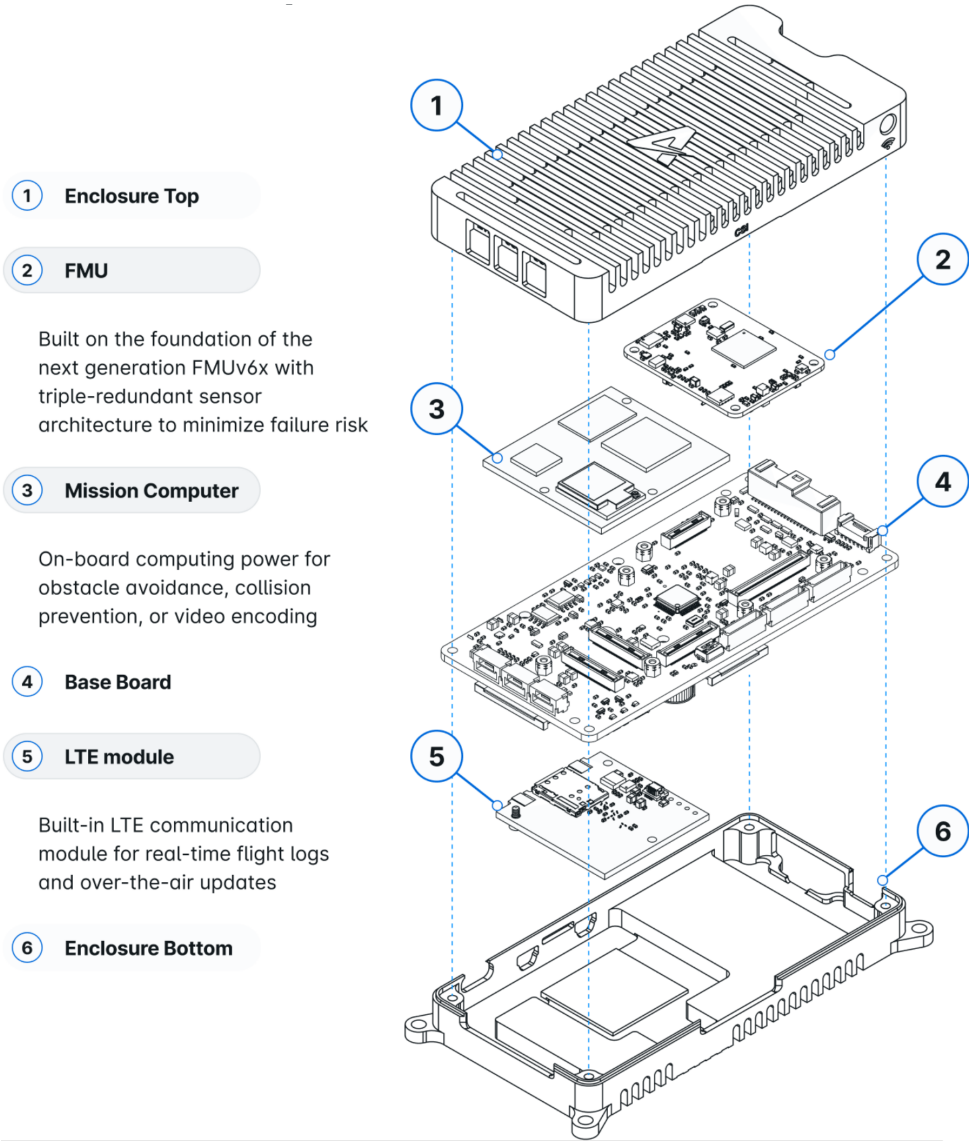


Figure 15: Auterion Skynode X

contains Ethernet and USB 2.0 high-speed interfaces only, as USB 3.0 usage is discouraged due to GPS interference. For video applications it provides Ethernet and USB connections with H.264 hardware encoding/decoding [76]. The mission computer runs Auterion OS (Linux-based), which communicates with the Autopilot via a proprietary SDK (Auterion SDK). It supports multicopter, VTOL airplane and airplane vehicles with a takeoff weight of up to 500 kilograms. The price tag is circa 1900 USD dollars[77].

Auterion also announced in June 2024, a compact, low-cost version – the Auterion Skynode S – integrating the FMU and mission computer in a 49 x 37 millimeters footprint [78]. It features the FMUv6x flight management unit from the current Skynode X family and a powerful mission computer with a dedicated 2.3 Trillion Operations Per Second (TOPS) Neural Processing Unit (NPU) for Artificial Intelligence (AI) and computer vision applications.

The AI capabilities of the Skynode S enabled the deployment of a autonomous target tracking system,

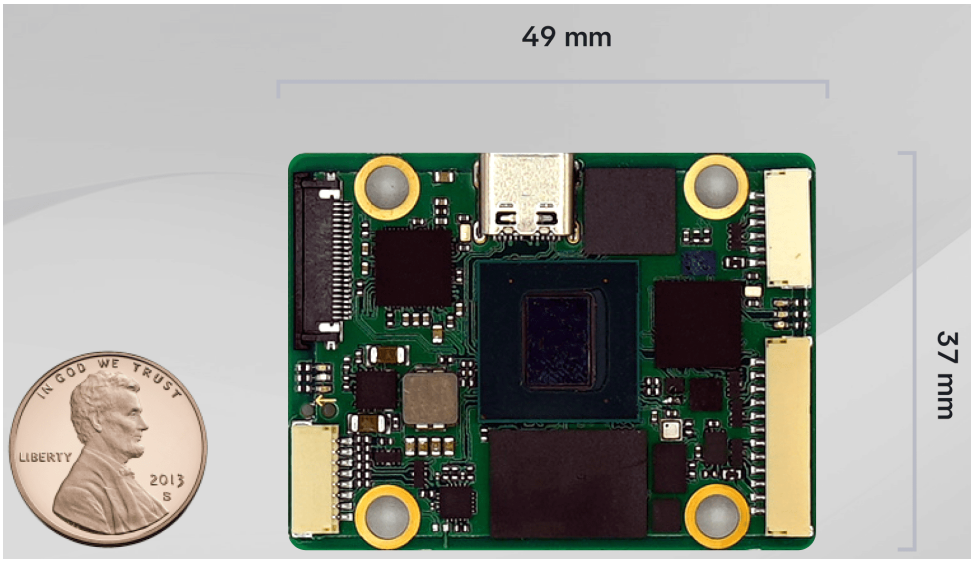


Figure 16: Auterion Skynode S

even for moving targets, which has proven its resilience against GPS jamming or denial of service [79]. This enabled Ukraine to face the Russian electronic warfare, and increasing the probability of mission's success from 20% to 90% [80].

2.2.5 UAV Reference Software

In this section the reference software for UAVs is discussed. An overview and the SW architecture is presented. Then, the open-source and commercial solutions are discussed, and compared in the gap-analysis.

2.2.5.1 Overview and Architecture

Fig. 17 illustrates the high-level abstraction of the most common structure of UAV SW architecture [55, 64].

The main computing platforms are depicted in orange – the flight controller (FCS), e.g., Pixhawk, the GCS (e.g., a desktop/laptop computer), and, optionally, the companion computer, e.g., a RaspberryPi. The companion computer provides additional features, but crucial for autonomous flights, such as collision avoidance and prevention, odometry or payload control (camera, LiDAR, etc.). Furthermore, the companion computer can assist in the UAV's navigation in GPS-denied and/or communications-denied environments, using AI-based software components (e.g., Auterion Skynode [79]).

The physical HW – sensors, actuators, and payload – is depicted in purple, and in green the layers of the SW stack. The arrows indicate the communication link and associated protocols. The SW stack structure is the analogous for the flight controller and the companion computer, and is comprised of four layers:

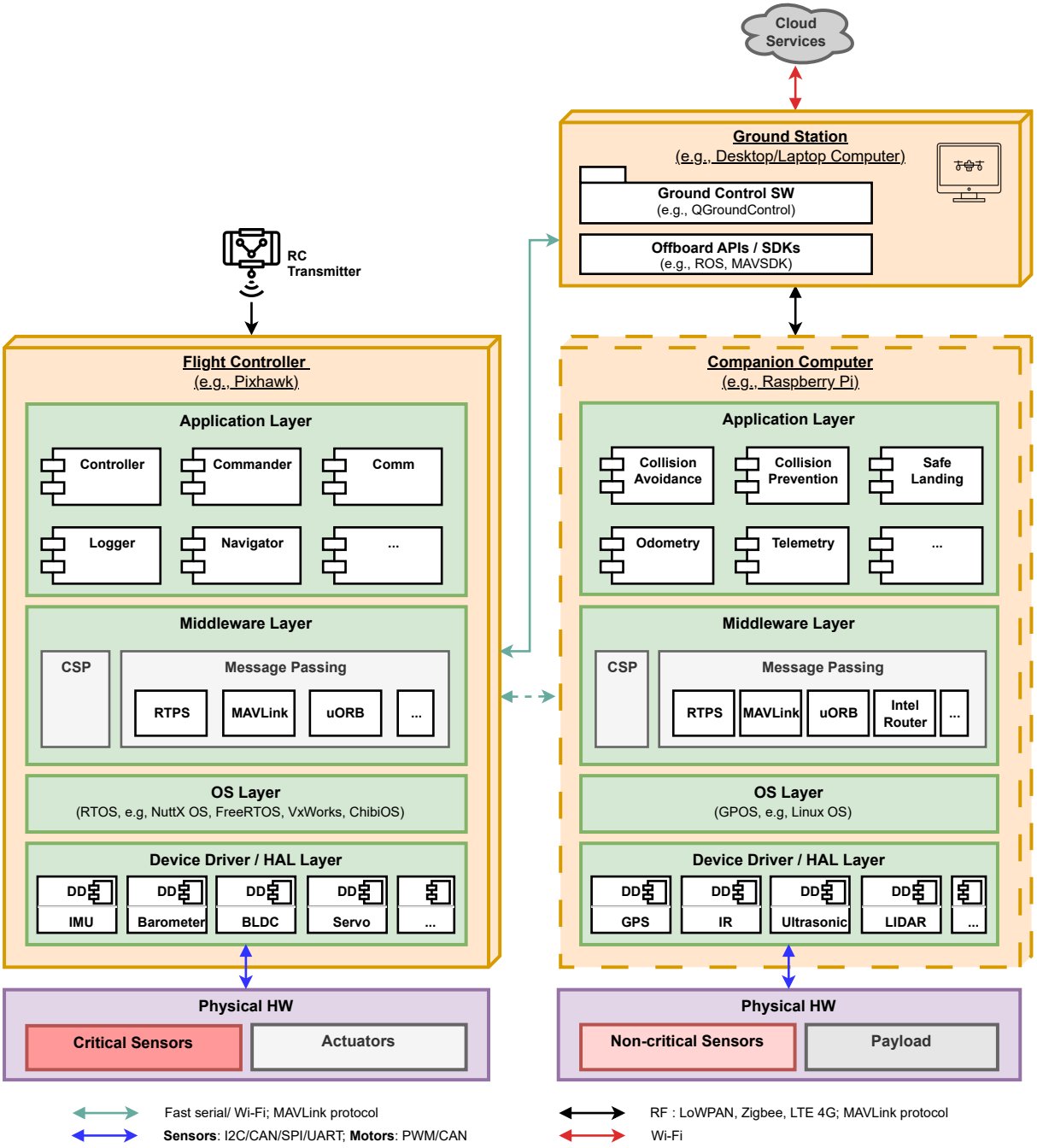


Figure 17: UAV SW architecture: most common structure

1. **Application Layer**: this layer is where the applications/tasks reside, providing the high-level functionalities of the system. In the flight controller we have the critical tasks, namely, *Controller*, *Commander*, *Navigator*, *Logger*, or *Communications*, and in the companion computer the secondary tasks, providing mission and navigation extra features, such as, *Collision Avoidance*, *Collision Prevention*, *Safe Landing*, *Odometry*, and *Telemetry*.

2. **Middleware Layer**: the middleware is an intermediate layer, responsible for abstracting the interface with the OS and the communications through message passing — using real-time and or

asynchronous protocols, such as Real-Time Publish Subscribe (RTPS), MAVLink, or Micro Object Request Broker (uORB) — and for providing an extra layer of security enforcing Context Security Policy (CSP) mechanisms. The flight controller communicates with the GCS through a Wi-Fi connection and optionally with the Companion Computer through a fast serial/Ethernet connection.

3. **OS Layer**: the OS provides the basic services for system's operation and a straightforward environment for application's development. The flight controller hosts a Real-Time Operating System (RTOS) to meet the stringent requirements and deadlines of the UAV's control, such as, `NuttX`, `FreeRTOS`, `VxWorks`, and `ChibiOS`. On the other hand, the companion computer hosts a General-Purpose Operating System (GPOS), as the applications on top have soft real-time requirements. Furthermore, it provides a much better bootstrapping environment for "general" software development than a RTOS, e.g., ROS-based avoidance libraries are available for Linux [64].

4. **Device Driver Layer (Hardware Abstraction Layer (HAL))**: this layer abstracts the underlying HW, providing a tractable interface for the OS for data acquisition and motors' actuation.

The GCS SW (e.g., *QGroundControl*) typically runs on a desktop/laptop computer or mobile device providing real-time monitoring of the flight superimposed on a map and UAV's control functionalities, communicating directly with the flight controller or indirectly via companion computer. The SW interface between both systems is achieved through the use of off-board APIs or SDKs, like ROS or MAVSDK [64].

2.2.5.2 Open-source solutions

Open-Source Software (OSS) solutions, like their HW counterparts, have been developed to provide more transparency, extensibility, flexibility, maintenability, while trying to be cost-effective. Open-Source Software (OSS) means that the SW's source code is released under free/libre terms [65].The three most prominent OSS solutions at the moment are `PX4`, `ArduPilot`, and `Paparazzi UAS`.

PX4 PX4 is an autopilot flight stack for drones, started in 2012, released under the permissible BSD-2 license, supporting different UAV's airframes and even others UVs. The source code core is on C/C++ and it is available on Github [81]. Several commercially available drones have adopted `PX4`, and a wider range supports it [82]. It supports NuttX and ROS operating systems [83].

PX4 is loaded onto the flight controller HW using the *QGroundControl* application. *QGroundControl* is also used to configure PX4 and interface the flight controller. A RC unit can be used to manually control the drone.

PX4 uses the MAVLink protocol to communicate with the flight controller, but also supports the RTPS protocol. It uses its message API — uORB — for data transfer between its internal modules. It is automatically started on bootup, and messages are defines as separate `.msg` files in the `msg/` folder [83]. PX4 uses several sensors for UAV's control or autonomous operation. The minimum set of sensors are

the accelerometers/gyroscopes, magnetometer and barometer, with a GPS being required for automatic modes. PX4 produces logs durint flights, which can be analyzed using *Flight Review*, or *Px4Tools* [3].

ArduPilot ArduPilot is an autopilot flight stack for drones, started in 2009, released under the more restrictable GPLv3 license, supporting different UAV's airframes and even others UVs. The source code core is on C/C++ and it is available on Github [84]. It supports *Linux* and *ChibiOS* operating systems [83].

It is installed on more than a million vehicles and runs on a large number of OSH platforms such as *Pixhawk*. *ArduPilot* features include: multiple flight modes (manual, semi- and full-autonomous) with multiple stabilization options; progammable missions with 3D waypoints and optional geofencing; support for multiple sensors and buses; fail-safe actions and support for navigation in GPS denied environments [85].

ArduPilot supports the MAVLink protocol for communication with GCSs and companion computers [85]. Mission commands are stored in Electrically Erasable Programmable Read-Only Memory (EEPROM) and executed one-by-one when the vehicle is switched into *Auto* mode.

Paparazzi UAS *Paparazzi UAS* is an autopilot flight stack for drones, started in 2003, released under the GPLv2 license, supporting different UAV's airframes UVs [86]. The source code core is on C (and a little C++) and it is available on Github [87]. It supports the *ChibiOS* operating system [83]. It was designed mainly for autopilot applications, for portability, and flexibility, with the ability to control multiple aircraft systems within the same system [86].

Paparazzi UAS uses its own *PPRZLINK* protocol for communication with GCSs and companion computers [83]. It uses a proprietary middleware Application Binary Interface (ABI) — *AirBorne Ivy* — to transfer data between modules in a straightforward way. Messages are defined in a eXtensible Markup Language (XML) file with an unique name and *id*. *Auto* mode.

Jargalsaikhan et al. [83] conducted a survey on the flight control software portability, more specifically at a module-level basis, analyzing several academia and OSS solutions features — general structure, messaging mechanisms, and supported OSs — and comparing them with the portability requirements — simplicity, moduarity, centralized messages, standard interfaces, HAL's implementation, and documentation. It found out that, from the aforementioned OSS solutions: only *ArduPilot* and *PX4* implement a HAL; *ArduPilot* does not implement a message centralization mechanism; although open-source, all use proprietary messaging interfaces, which hinders portability; and that although well documented, sometimes is hard to understand and/or is partially outdated. Fig. 18 illustrates the SW stack layers and modules for the abovementioned OSS solutions and the *mpFCS* — the portable FCS developed by Jargalsaikhan et al. [83].

[4]Used with permission from MDPI: license nr. 5457890117132

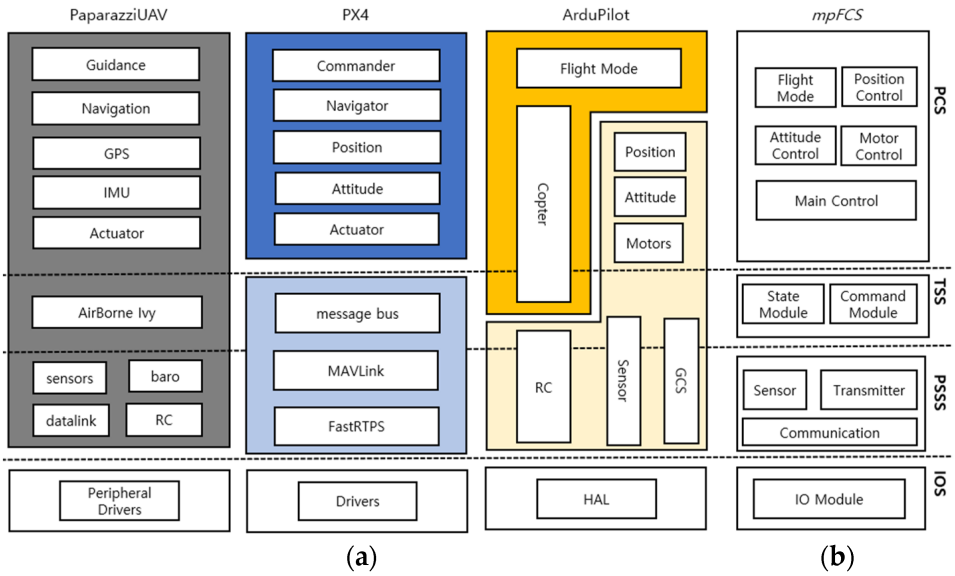


Figure 18: Analysis of the OSS modules and the mpFCS: (a) OSS (Paparazzi UAS, PX4, ArduPilot); (b) mpFCS [83][4]

2.2.5.3 Commercial solutions

Typically, the commercial proprietary solutions do not disclose information about the product. They provide extensibility through the use of SDKs, like the Dji drones [45], enabling the end-user to use extra features, but requires specialized knowledge.

On the other hand, and as aforementioned, some commercial solutions can explore more permissible OSS licenses, like the BSD license provided by PX4, to add extra functionalities and bundle it as a closed, commercial, product. This is exactly the case for the Auterion PX4 (PX4) autopilot, a enterprise-hardened autopilot that runs on the Auterion FMUs. Fig. 19 illustrates the Auterion software stack.

The `Auterion Skynode` combines the flight controller – a FMUv6X-based Pixhawk FMU and the companion/mission computer into a single board [88]. The flight controller runs the autopilot stack – APX4 – on top of the NuttX RTOS licensed under the Apache 2.0 license which allows for proprietary reuse. The mission computer runs the Auterion OS (AOS), a customized embedded Linux OS, to support additional features, like collision avoidance and prevention, odometry, or camera control [88]. These additional features or services, are containerized in a Docker image to increase isolation and enable easy and fast system update, minimizing the dependencies. However, to optimize storage requirements, developers are instructed to: (1) package these services into a single application (e.g., `Companion App`), ensuring the user only needs to handle a single `.auterionos` file; (2) write a common base `Dockerfile` for a shared environment for the deployed services [89]. Thus, AOS software development follows the microservices architecture based on the Docker technology. The AOS' applications can query the autopilot or issue commands to it using the provided APIs in the `Auterion SDK`.

The `Auterion Skynode` can be controlled using a RC transmitter or software running on the GCS. The `Auterion Mission Control` [90] is a ground control software specialized for Auterion vehicles, very similar to `QGroundControl`, enabling the UAV's remote control. The `Auterion Suite` [91] is a

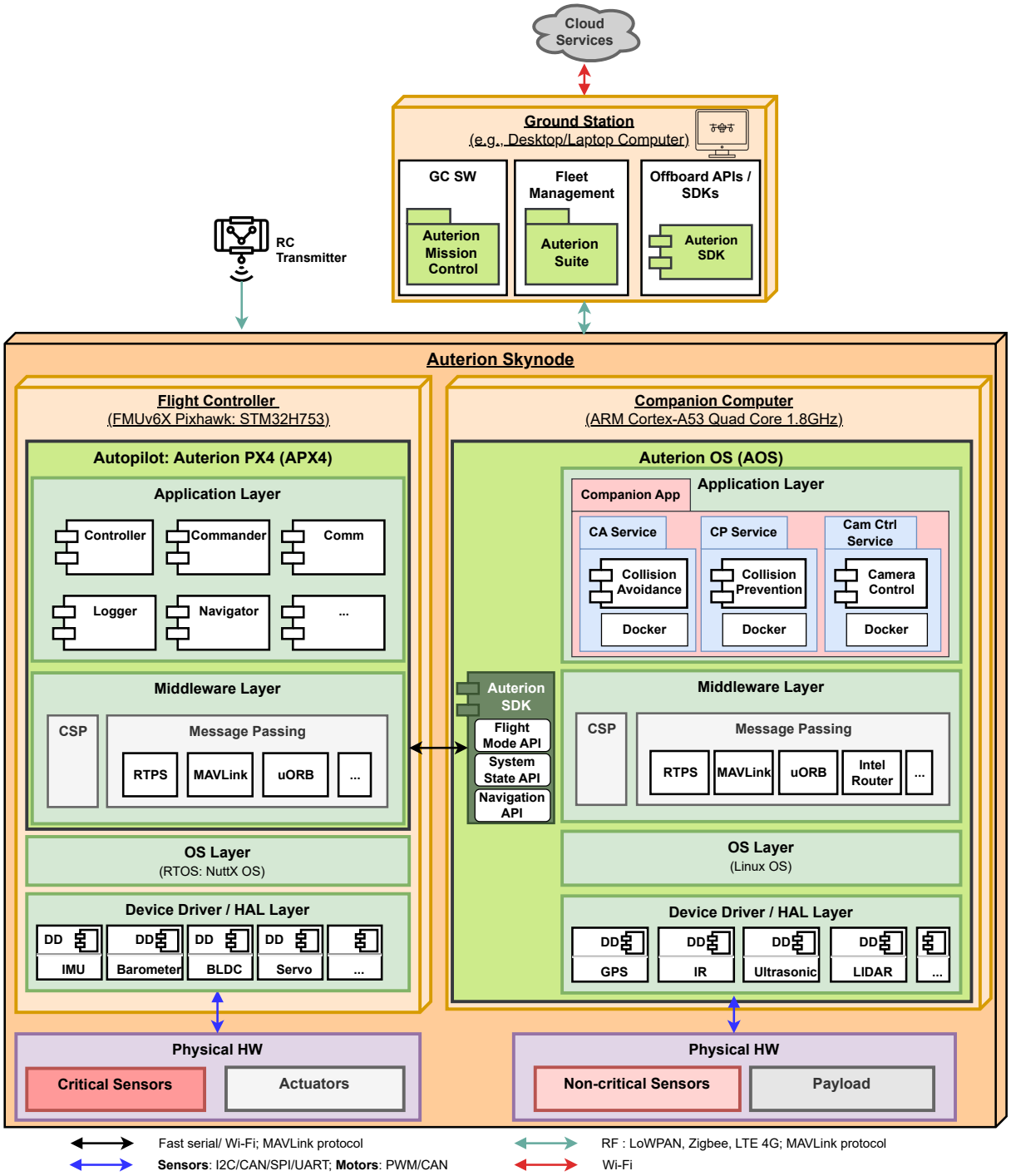


Figure 19: UAV SW architecture: Auterion software stack

fleet management SW, providing real-time information about the UAV, predictive maintenance actions, flight analysis, and Over-The-Air (OTA) updates.

If, however, we expand the search to include any proprietary software component, such as the `VxWorks` RTOS then the results' list becomes significantly larger, namely, the Northrot Grumman X-47B UAV [92], the Airbus Helionix [93], and the Airbus Atlante [94].

On academia, *VxWorks* was also used as RTOS for UAV applications. Chong and Li [95] designed

a FCS for small UAVs using *VxWorks* as an RTOS to support tasks partitioning meeting the real-time requirements. Fig. 20 illustrates the software framework designed considering three main layers: functional layer (application + middleware), OS layer, and device driver layer. The authors claimed remarkable improvements in the reliability and real-time capability of the FCS using `VxWorks`, due to its multitasking capabilities, but did not provide direct comparison with other RTOSs.

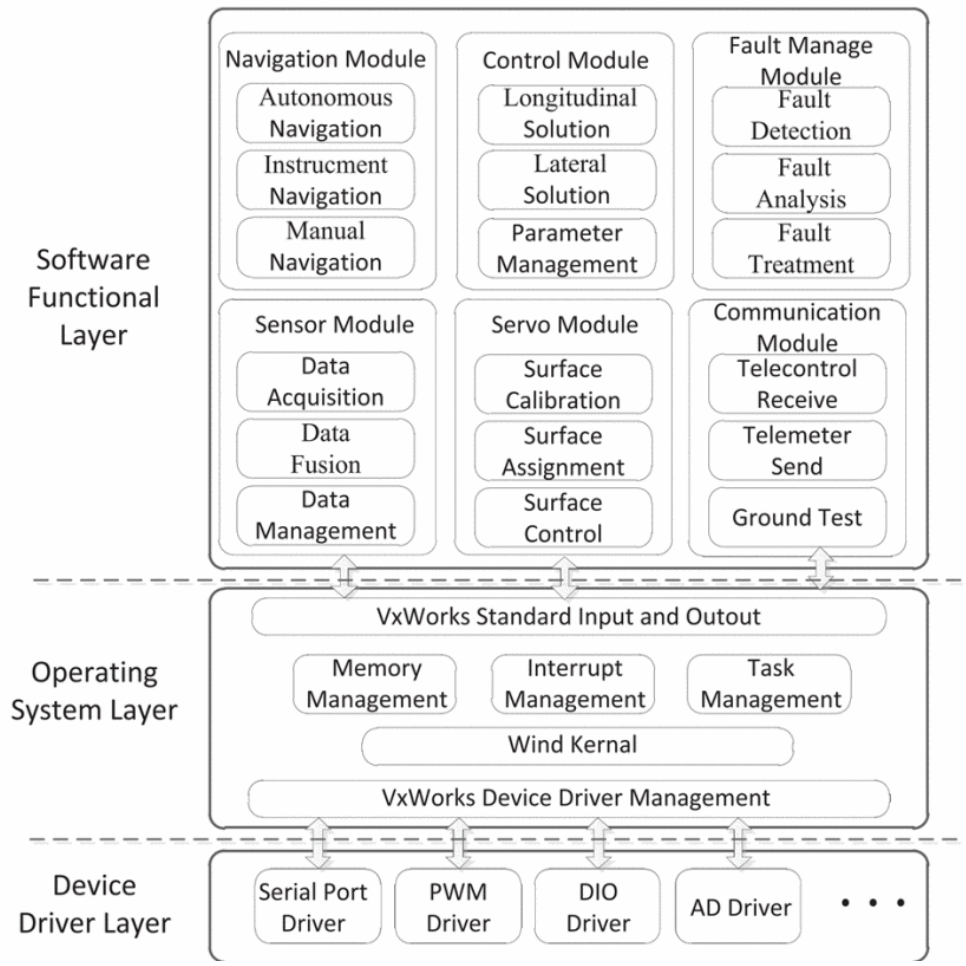


Figure 20: SW framework for an UAV application based on the VxWorks RTOS [95][5]

2.3 Related work

UAVs's market is booming, but the security and safety is often overlooked in the design [63]. Several vulnerabilities have been identified [4, 5, 56], with a extensive attack surface area — FCS, GCS, chassis, FPV channel, and the cloud services.

The commercial SW solutions do not provide the security mechanisms required, and, furthermore lack transparency. Buquerin [96] conducted a security evaluation on the VxWorks 7 RTOS — a commercial professional-level RTOS — for avionic systems. The author reported that basic attacks such as buffer

[5]Used with permission from IEEE: license nr. 5457890117132

overflows or string vulnerabilities led to a noticeable performance decrease in the system. Moreover, no security mechanism protects the system from malware or command injection vulnerabilities. This poses a major threat on the system if no supervising technology, like virtualization, is used. On the other hand, good security practices are used in areas such as cryptography and privilege management. To mitigate this issue, Auterion employs in the mission computer a microservices architecture based on application containerization using the Docker technology [89]. However, as all applications share the same host OS kernel, a misconfigured or ill-behaving application can compromise the mission computer, and inject erroneous commands into the autopilot, compromising the whole system. Furthermore, the autopilot runs unsupervised on the flight controller, i.e., it typically has full access to all of the hardware resources, increasing the attack surface area. Executing the autopilot in supervised mode, namely with static partioning of resources, can mitigate this issue by enforcing the User to carefully select the appropriate hardware for each mission, but also by trapping, and thus preventing, any ill-intentioned access to unavailable hardware resources.

With the open-source solutions, comes transparency, but, nonetheless, it is insufficient to meet the mixed-criticality and security requirements. Zhang et al. [97] compared the two main RTOSs for drone's applications in the open-source domain, Nuttx and ChibiOS, with the latter emerging as the best. Not only ChibiOS outperforms NuttX in most temporal tests, but also, and even more importantly, succeeds in the priority inversion avoidance test with mutexes rather than binary semaphores. The context switch overhead is greater for the NuttX, augmenting the probability of missing deadlines. Furthermore, NuttX does not provide priority inheritance using mutexes, which may lead to faulty designs, and has a larger codebase. Since UAVs have stringent real-time constraints, ChibiOS appears to the be the best RTOS for FCSs, and therefore the migration of ArduPilot's from NuttX to ChibiOS is probably a smart decision.

This leads to some research being conducted on changing the paradigm entirely. Alladi et al. [2] propose a completely different approach — the use of a decentralized architecture to increase security in UAVs application through the application of blockchain — especially in cooperative environments. Blockchain features such as smart contracts and consensus mechanisms and the inherent cryptographic foundations, enabling automation and providing security, according to the authors. Overall, the blockchain technology helps in overcoming many problems such as coordination, security, collision avoidance, privacy, decision making, and signal jamming. Although it presents great potential, the real-life implementation seems a little distant.

On the other hand, virtualization is a well-proven and mature technology, emerging as a viable solution to handle the UAV's mixed-criticality requirements and security, through spatial, temporal, and fault isolation. Nonetheless, the number of works in these field is still scarce. Faultrel et al [98] proposed an hypervisor based approach for mixed critical real-time UAV applications to meet its stringent timing and criticality requirements and allow FCS certification. The authors used an iterative approach for the construction of scheduling tables at the hypervisor level. This approach was then applied to the use case of an industrial project of inspection drone systems comprising three VMs of different criticality levels: (1) high – autopilot, running on baremetal; (2) medium – communication application between the GCS and

the UAV, running on OpenWRT; (3) low – video application that stores, analyzes, and compresses images, running on a Linux Debian OS. The authors used the PikeOS level 1 hypervisor as it can be certified DO 178B/C for flying systems. However, PikeOS is closed source, hindering the widespread adoption to small and inexpensive UAV systems.

It is important to note that virtualization is also broadly used in the literature to encompass the usage of UAV in larger, adaptable networks, where each UAV can provide different services depending on a dynamic configuration. This is also referred to as softwarization of UAV networks [99]. For example, Nogales et. al [100] proposed the deployment of small UAVs capable of executing virtual functions and services for rapid adaptation to different missions with heterogeneous objectives. The authors implemented an Internet Protocol (IP) telephony service as a set of virtualized network functions, tested with real voice-over-IP terminals.

2.4 Summary

UAV are being extensively deployed in a myriad of applications, such as: search and rescue missions, agriculture and farming, military missions, delivering goods and medical supplies, video capturing and filming, providing telecommunications in remote areas, among others. Nonetheless, vehicle's security and safety is often overlooked, with several documented vulnerabilities and an extensive attack surface area.

The UAV architecture is typically heterogeneous by nature and with different criticality levels, i.e., a Mixed-Criticality System (MCS), comprising a critical and a non-critical system. The critical system runs the autopilot software stack in the flight controller to control the UAV, sampling the critical sensors (IMU, gyroscope, compass, and barometer) and computing the required actuator values to maintain the navigation heading according to the specified parameters. The non-critical system provides additional features such as collision detection and avoidance or payload control. It uses a so called companion or mission computer for this purpose. Although optional, the companion computer is crucial for autonomous flights, and can be enhanced to provide AI-assisted navigation in GPS-denied or communications-denied environments. A RC transmitter can be used to manually operate the UAV, while the companion computer can command the flight controller to follow several waypoints in mission mode, typically using a fast serial link. The ground station runs the Ground Control Station (GCS) to assist in the mission planning, but also for configuring the UAV and to obtain real-time flight data via telemetry radio or LTE link. Typically, the UAV is battery-powered, but fuel or hybrid systems are also common. The most notable open-source flight controllers are the Pixhawk (microcontroller-based) platforms, while the commercial alternatives range from the microcontroller-, FPGA-, or companion-based platforms. Perhaps, the most notable of the commercial solutions is the Auterion Skynode X based on the Pixhawk FMUv6 architecture (BSD license). It combines a flight controller, a mission computer and LTE connectivity in a compact form factor.

Regarding the software stack, the flight controller and companion computer are structured similarly in

four layers: application, middleware, OS, and HAL. The application layer provides the high-level functionality of the system, the middleware abstracts the interface with the OS and the communications through message passing, the OS provides the basic services for system's operation, and the HAL abstracts the underlying HW. The flight controller runs the autopilot (application + middleware) (e.g., PX4 or Ardupilot) on top of a RTOS (e.g., NuttX, ChibiOS, VXWorks). The mission computer runs a GPOS, typically Linux-based. On the commercial side, Auterion runs an enterprise-hardened version of PX4 (APX4) on the FMU, and a customized version of an embedded Linux OS (AOS). It provides an SDK with the relevant APIs to enable the AOS to query and command the APX4. It provides a microservices architecture based on the Docker technology to deploy applications on top of AOS.

The commercial SW solutions lack security mechanisms, rendering it susceptible to basic attacks such as buffer overflows or strings vulnerabilities, or more advanced ones such as malware or command injection. To mitigate this issue, Auterion employs application containerization in the mission computer. However, as all applications share the same host OS kernel, a misconfigured or ill-behaving application can compromise the mission computer, and inject erroneous commands into the autopilot, compromising the whole system. Furthermore, the autopilot runs unsupervised on the flight controller, i.e., it typically has full access to all of the hardware resources, increasing the attack surface area. Executing the autopilot in supervised mode, namely with static partioning of resources, can mitigate this issue by enforcing the User to carefully select the appropriate hardware for each mission, but also by trapping, and thus preventing, any ill-intentioned access to unavailable hardware resources.

Due to the wide and varied attack surface, some research focuses on changing the paradigm, e.g., using a blockchain decentralized architecture to mitigate many UAV problems such as coordination, security, collision avoidance, privacy, decision making, and signal jamming. Although it presents great potential, the real-life implementation seems a little distant.

On the other hand, virtualization is a well-proven and mature technology, emerging as a viable solution to handle the UAV's mixed-criticality requirements and security, through spatial, temporal, and fault isolation. Nonetheless, the number of works in these field is still scarce and are based on closed source hypervisors, such as PikeOS.

Thus, the research demonstrates the UAV software stack can benefit from the usage of virtualization to handle mixed-criticality requirements in a secure way. This is especially true for the open-source software stack. Virtualization provides isolation and can assist in the consolidation of the computing platforms, minimizing the SWaP-C metrics.

3

Design

"*Simplicity is the ultimate sophistication.*"
– Leonardo Da Vinci, polymath

Addressing UAVs' conflicting demands for security, safety, and SWaP-C efficiency necessitates rethinking conventional mixed-criticality architectures. This chapter presents the design of the SSPFS: a trustworthy flight stack leveraging the Bao hypervisor for video surveillance applications, where resource-intensive but non-critical streaming coexists with safety-critical flight control on consolidated hardware.

The conventional UMPFS approach employs separate hardware nodes—a flight controller for critical systems and a companion computer for tasks like collision avoidance or video streaming. While providing functional separation, this architecture increases weight and power consumption while offering no isolation guarantees, allowing compromises in the companion node to propagate to flight systems.

Merging flight control and companion functions onto a single platform reduces weight, power requirements, and inter-component latency. However, unsupervised consolidation (USPFS) introduces critical risks including performance interference (where non-critical tasks disrupt flight-critical operations) and security vulnerabilities (where a single compromise affects the entire system). Consequently, unsupervised consolidation alone cannot meet stringent safety and security requirements.

The supervised approach hypothesizes that hypervisor oversight enables secure consolidation while preserving SWaP advantages. The SSPFS employs the Bao hypervisor to achieve hardware-enforced isolation between flight control and companion functions, certifiable separation with minimal performance overhead, and retention of single-board SWaP benefits. The following sections detail this architecture's requirements, components, and security model.

3.1 Requirements and Constraints

Video surveillance missions necessitate geolocation control to survey a designated target area and image acquisition to gather pertinent information about that region. Both objectives can be achieved through offline or online command methods, or a combination of the two.

In the offline command approach, the target area and the specific information to be captured are well-defined and can be comprehensively specified *a priori* to the UAV. For instance, in cartographic applications, the UAV systematically scans the target area to collect topographic data. Consequently, the UAV can operate in a fully autonomous mode, with data being directly stored in its onboard storage systems.

Conversely, the online command approach is more suitable for dynamic and unpredictable environments, where the target area and relevant information are not fully predefined. For example, in rescue missions, the identification of targets is critical, necessitating active supervision by the GCS. In this context, the GCS must have the capability to remotely control the UAV and receive real-time feedback, such as telemetry data and live video streams.

This example underscores the critical importance of the online command method, which involves more stringent operational requirements. As a result, the primary focus of this work will be on the online command approach.

Table 3 lists the requirements and constraints for the UAV flight stack. Functional requirements include real-time telemetry and command capability, video surveillance, autonomous flight control, and battery operation. Technical requirements mandate fault-tolerant isolation between components, minimal security overhead, and hardware consolidation of flight control and companion functions. Functional constraints prioritize weight minimization and bandwidth-optimized video transmission, while technical constraints specify an open-source software stack, wireless communications, and the use of the Bao hypervisor for security enforcement.

Table 3: System requirements and constraints for the UAV flight stack

Categorization	**Functional**	**Technical**
Requirements	• Remote UAV command/geolocation in soft real-time • Real-time image capture capability • Onboard flight stabilization mechanisms • Battery-powered flight autonomy	• Fault tolerance: Video compromise must not affect flight control • Minimal latency impact from security mechanisms • Consolidated flight/companion functions on single hardware
Constraints	• Minimal weight for flight autonomy • Bandwidth-optimized video transmission	• Open-source flight stack (e.g., PX4) • Wireless control/image transmission • Bao hypervisor for security/fault-tolerance

3.2 System Architecture

Fig. 21 illustrates the conventional solution — Unsupervised Multi-Platform Flight Stack (UMPFS) — tailored for the video surveillance application, which employs a separation of concerns. In this approach, the flight controller hardware node manages flight-critical systems, while the companion computer hardware node handles secondary and computationally intensive tasks, such as collision avoidance and prevention, odometry, or, in this case, video streaming to the GCS.

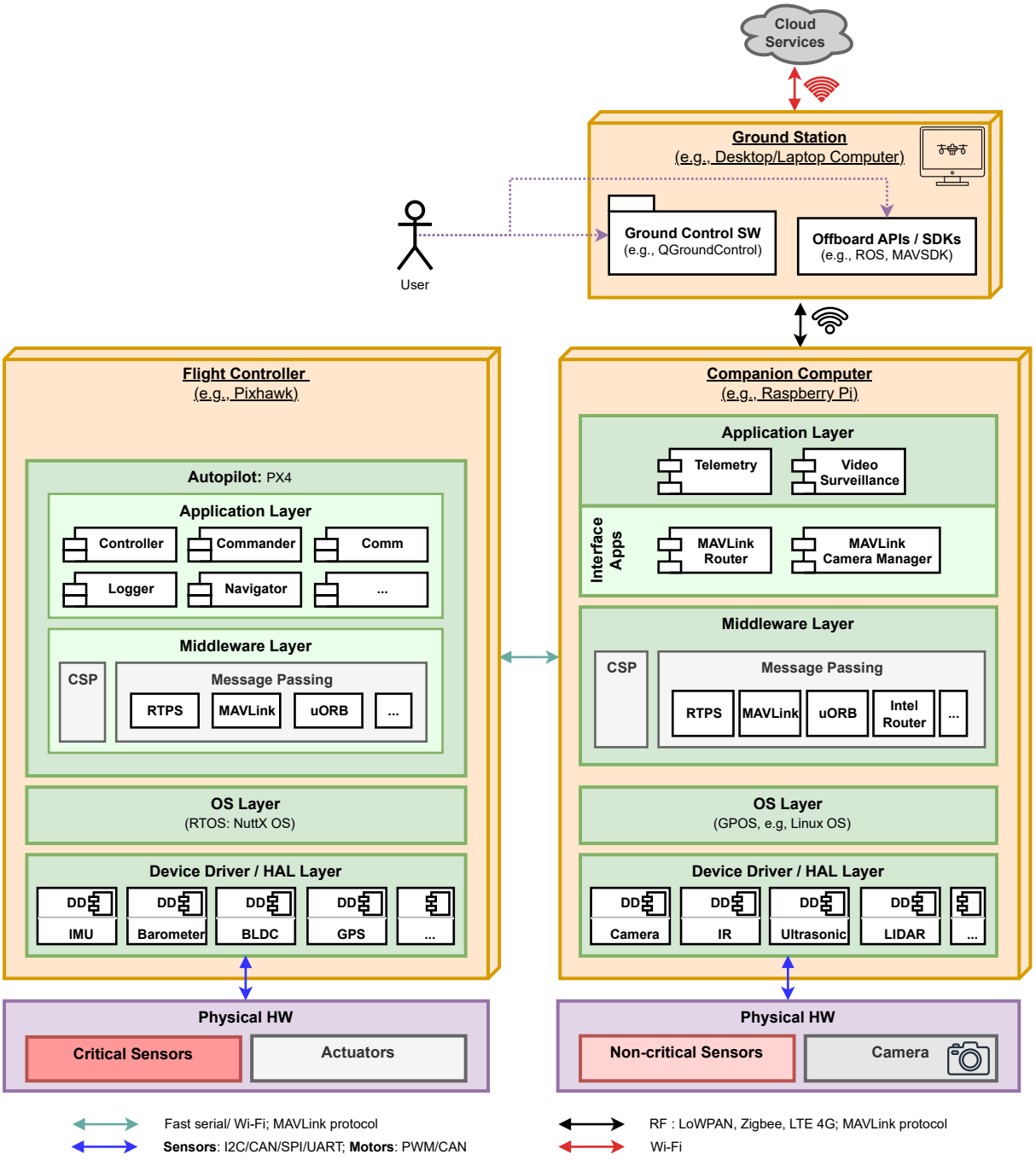


Figure 21: UAV design: conventional solution — full

The PX4 flight controller stack was selected for this work due to its open-source nature, extensive platform support, modular architecture, and widespread adoption in the industry. PX4 runs on the flight controller on top of the NuttX RTOS. In the generic case, the companion computer is used to route communications between the GCS and the FMU: a fast serial link, typically UART or Ethernet, is established between the FMU and the companion computer using the MAVLink protocol; the companion computer runs extra software to route the MAVLink traffic, e.g. MAVLink Router [101].

The `User` interacts with the Ground Control SW, namely `QGroundControl`. `QGroundControl` was selected due to its open-source nature and compatibility with the PX4 flight stack. Additionally, the User can interact with the Companion Computer via offboard APIs/SDKs, e.g., `MAVSDK`. The RC link was dropped, as it is only usable in manual mode, thus, it is not useful for the video surveillance application.

In the vast majority of cases, extra software, running on the companion computer, is required to interface the camera, e.g. the Mavlink Camera Manager. This component acts as bridge between the FMU and GCS and a translator between the MAVLink Camera Protocol v2 (used by PX4) and the native protocol of the camera [102].

The communications routing poses an increased risk to the UAV: if the companion computer is compromised the FMU may malfunction due to data corruption or communication loss. Furthermore, the additional software required to route MAVLink traffic and manage the camera adds complexity and latency to the system.

Fig. 22 showcases a simplified conventional solution customized for video surveillance, with a higher degree of decoupling. Dedicated communication links are established for communication between the GCS and FMU (telemetry radio) and the camera (Wi-Fi). To streamline the network configuration and eliminate the need for additional hardware, the GCS and the UAV are integrated into the same Local Area Network (LAN). The video surveillance software is also simplified consisting of a client running on the GCS and the server running on the companion computer supported by a suitable video pipeline and device drivers. The client runs on port 5000 of the GCS issuing command for the server running on the same port in the Companion Computer. The server handles commands and requests the sender to setup the video pipeline and transmit video frames back to the GCS. The receiver sets up a video pipeline on the GCS (not displayed) that processes the video frames and displays it to the `User`.

On the receiving end of the video surveillance system (GCS), occasional frame loss is tolerable and does not compromise situational awareness of the target area. Consequently, a communication protocol without delivery guarantees, such as UDP, is appropriate. This simplified conventional solution forms the base design for the platform unification.

3.2.1 Unsupervised Single-Platform Flight Stack

Integration of the FMU and companion computer platforms requires encapsulation of their functionalities into standalone components within a unified platform. In the Unsupervised Single-Platform Flight Stack (USPFS), these components are abstracted as processes running on a GPOS.

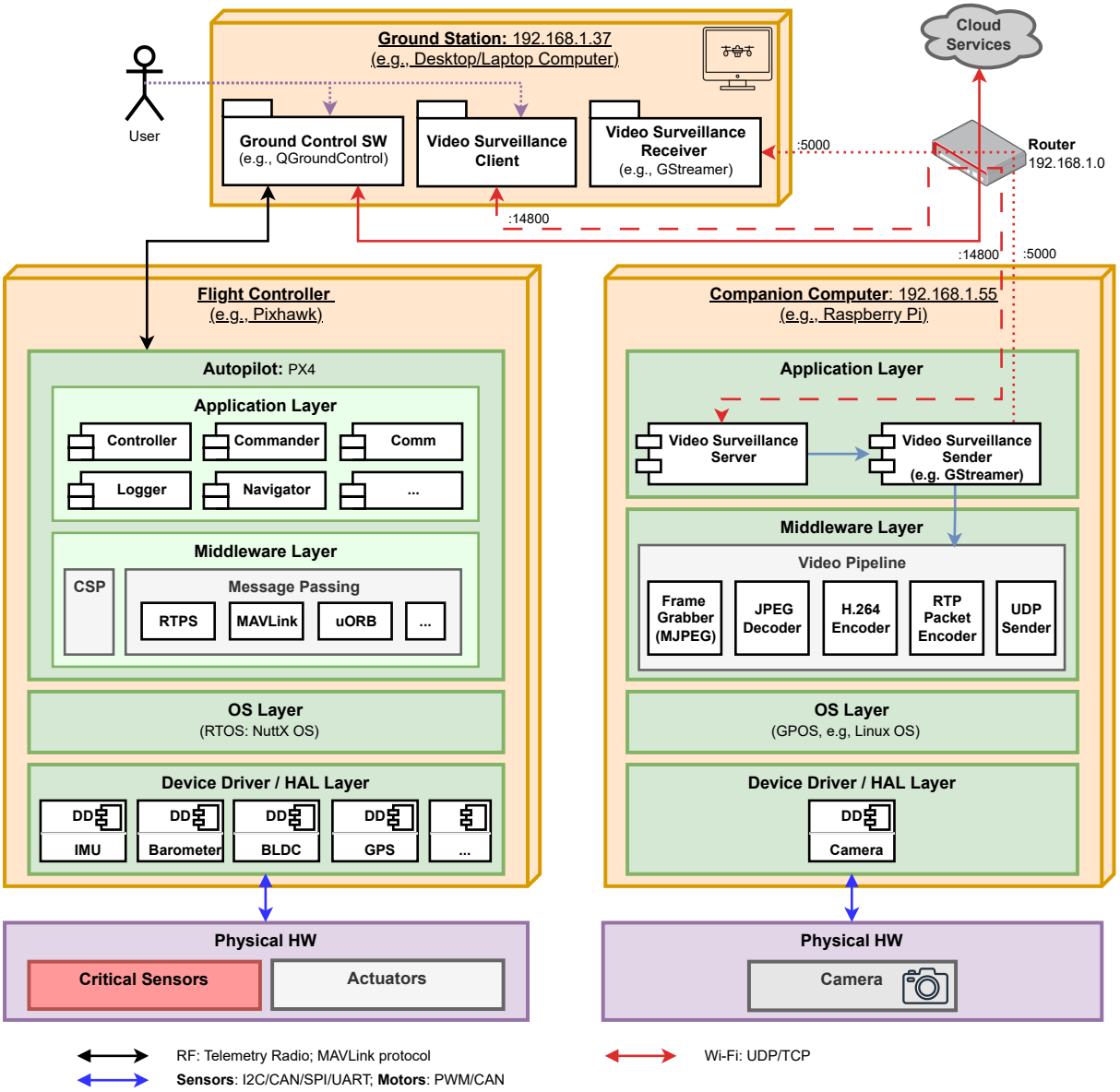


Figure 22: UAV design: conventional solution – simplified

Fig. 23 illustrates the USPFS system architecture. Software processes for the GCS and UAVIC appear in blue. The UAVIC consolidates both FMU and companion computer nodes onto a single platform operating on a GPOS (specifically Linux), with PX4 executing on core 0 and remaining cores allocated to the video surveillance application.

It is important to note that PX4 no longer runs on the Nuttx RTOS, which may introduce challenges in meeting the soft real-time requirements of flight control. To address this, one core is explicitly dedicated to the PX4 application. Additionally, employing a real-time Linux kernel with a suitable I/O scheduler can help mitigate timing issues.

However, this architecture lacks isolation between systems, meaning a failure in the non-critical system can propagate to the FMU. Such failures could lead to FMU malfunctions, potentially resulting in a crash

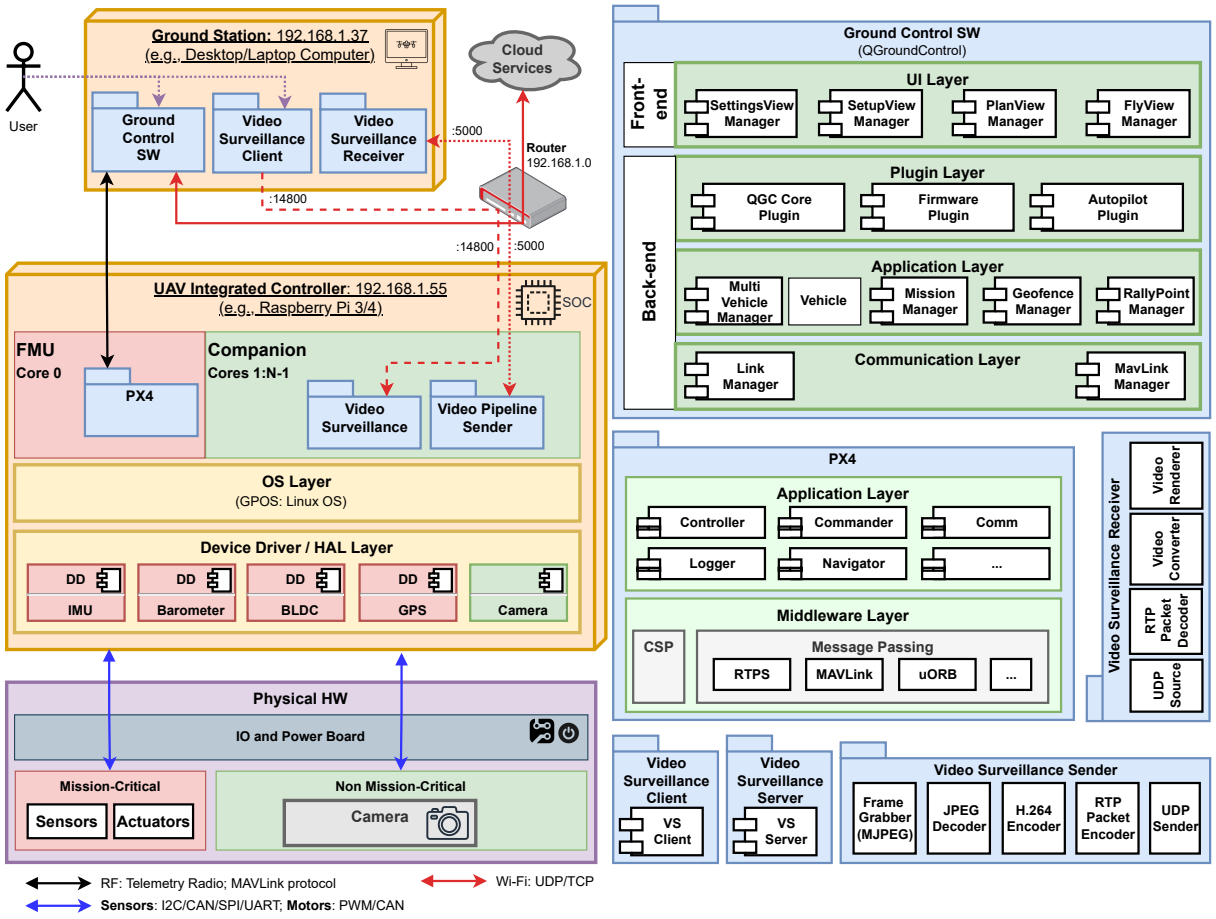


Figure 23: UAV design: Unsupervised Single-Platform Flight Stack

with unpredictable consequences. As mentioned earlier, this solution alone is insufficient; supervision is essential to ensure both reliable consolidation and safe integration.

3.2.2 Supervised Single-Platform Flight Stack

Fig. 23 presents the system architecture of the Supervised Single-Platform Flight Stack (SSPFS). In this design, the functionalities of the FMU and companion computer are abstracted as guest VMs running atop the Bao Hypervisor in the UAVIC node. This approach ensures isolation between the two mixed-criticality systems, preventing faults in the non-critical system from impacting the FMU and causing potential malfunctions.

Each VM operates a Linux-based OS, enabling further customization. For instance, the FMU VM can utilize a real-time kernel, providing additional guarantees that its assigned core remains fully dedicated to its execution. Meanwhile, the Companion VM can operate with a standard Linux kernel. Bao's static partitioning mechanism guarantees that the hardware resources assigned to each VM are strictly dedicated, ensuring isolation. Moreover, device drivers within each VM are specific to that VM, minimizing the impact of software bugs or failures.

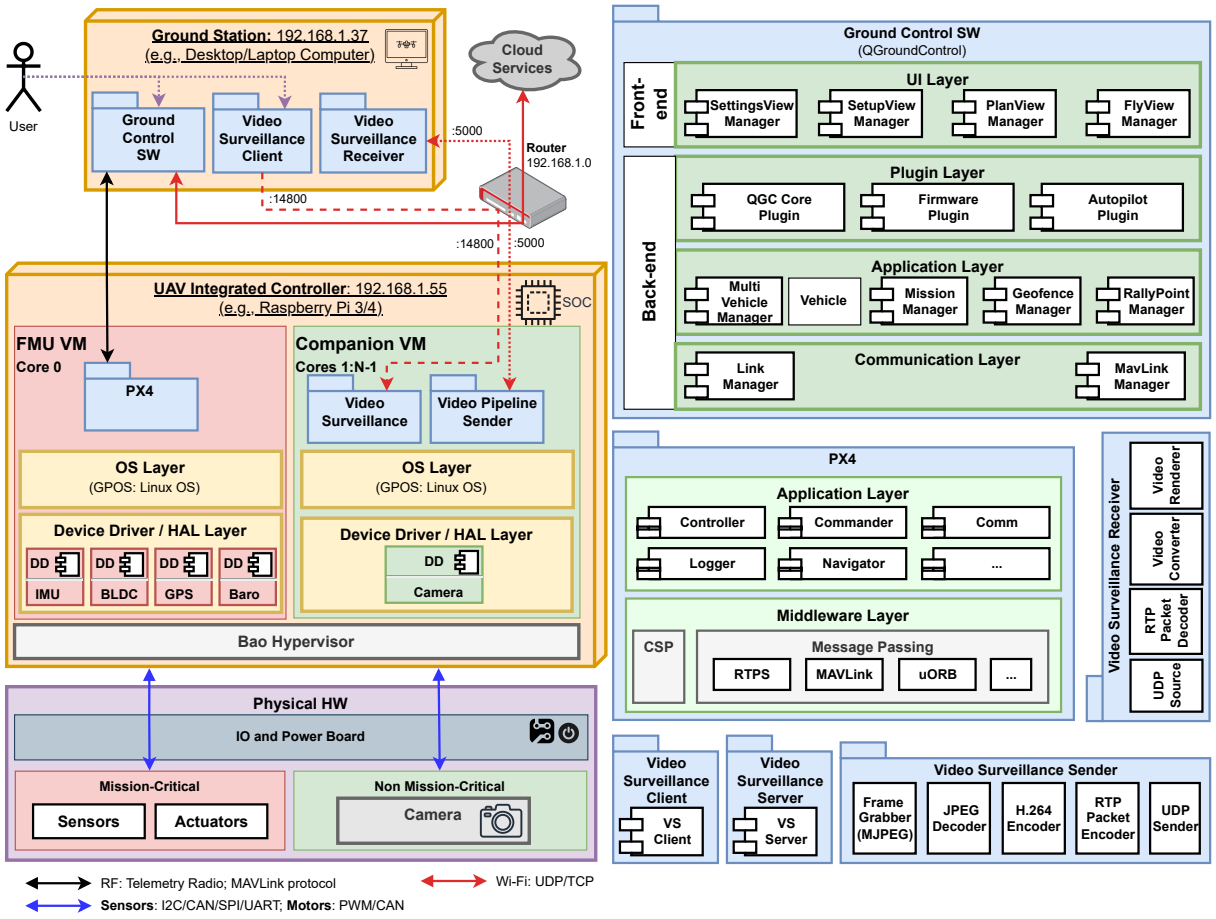


Figure 24: UAV design: Supervised Single-Platform Flight Stack

However, this approach requires each VM to include a full OS, leading to larger binary sizes. Additionally, the `User` must carefully select and allocate hardware resources to avoid conflicts between VMs while maintaining adequate performance in terms of RAM, available CPUs, and other critical resources. This requires a deeper knowledge about the hardware used. As such, the system architecture requires later adaptation to the selected hardware.

3.3 Hardware Selection

In this section, the hardware for the UAV, UAVIC, and extra addons are selected. The hardware is mapped to comply with PX4 requirements and the UAVIC restrictions imposed by Bao.

3.3.1 UAV

The UAV's main characteristics are defined as follows. The multirotor airframe selection prioritizes low cost, high availability, compactness, and VTOL capability. Operation occurs within the LAP altitude range (3-9 kilometers). The design requires battery power with minimized weight to extend flight time. Crucially,

an open-source architecture enables user modification and customization.

Fig. 25 shows the selected UAV — the KIT-HGDRONEK66, commonly referred to as the NXP HoverGames UAV kit [103]. This Do It Yourself (DIY) professional development kit, priced under 500 USD, features the RDDRONE-FMUK66 as the FMU unit (1).

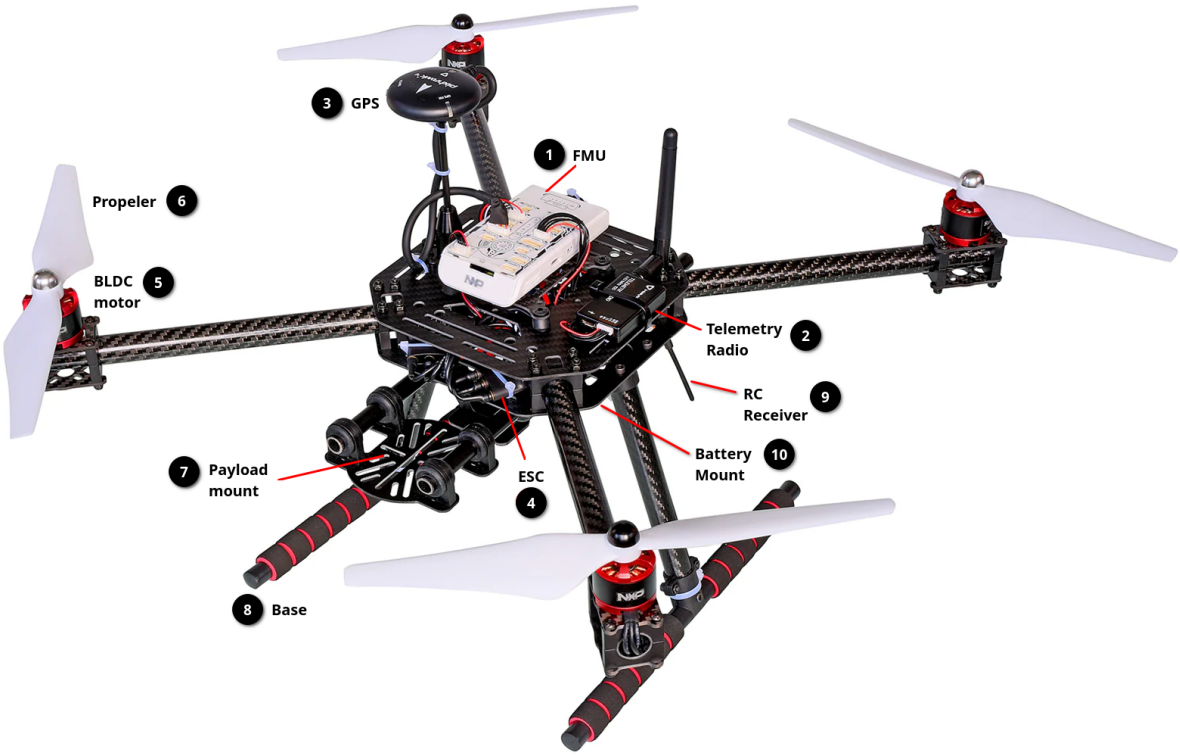


Figure 25: NXP HoverGames UAV kit (adapted from [103])[1]

The kit is built on an S500 carbon fiber frame with four rotors (quadcopter) and has a 500-millimeter wheelbase (diagonal distance between opposing motors). It employs BLDC motors (5) to drive the propellers (4), which are controlled by individual ESC units (4).

For autonomous flight capabilities, the kit includes a GPS module (3) and a payload mount (7) designed for accessories such as cameras. Power is supplied via a 3S LiPo battery (sold separately) with a capacity ranging from 3500 to 5000 mAh.

Additionally, a telemetry radio (2), also sold separately, can be connected to the FMU (1). This telemetry module operates in the 433 MHz frequency band in Europe, enabling remote communication and monitoring. The kit also includes the RC remote controller GS-i6S transmitter and receiver modules.

Fig. 26 depicts the block diagram for the NXP HoverGames UAV kit. It features the RDDRONE-FMUK66 FMU using the Kinetis®K66 MCU based on the 32-bit Arm®Cortex®–M4 Core [104], running at 180 MHz with up to 2 MB of flash and 256 KB of SRAM. The FMU runs the PX4 autopilot stack.

[1]Copyright © NXP Semiconductors: Used with permission for non-commercial purposes only
[2]Copyright © NXP Semiconductors: Used with permission for non-commercial purposes only

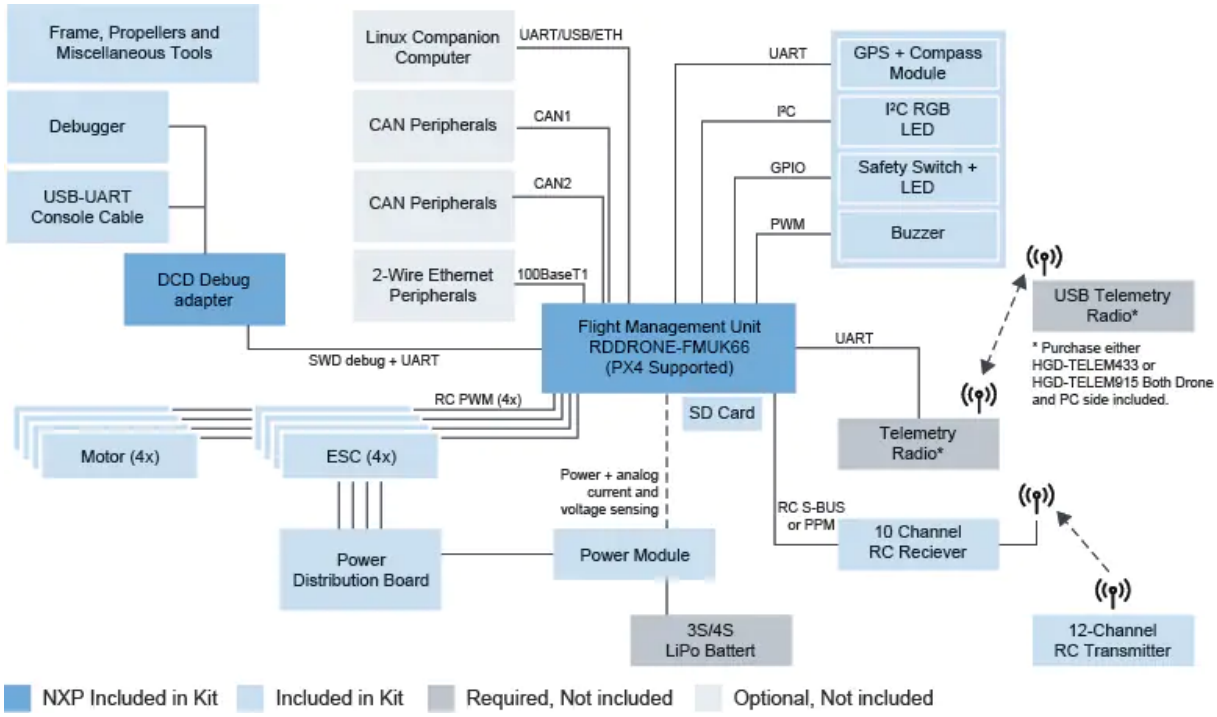


Figure 26: NXP HoverGames block diagram (withdrawn from [103])[2]

It includes a power module and power distribution board with current and voltage sensing to assess the UAV's LiPo battery autonomy. The power distribution board supplies the BLDC motors, controlled by the ESC unit using PWM. A SEGGER J-Link EDU Mini Serial Wire Debug (SWD) adapter can be used to debug the running flight stack or to upload it to the FMU.

It supports common UAV sensors such as accelerometer, gyrometer, magnometer, compass, and barometer. The FMU can communicate with a companion computer via UART, or with the GCS via the telemetry radio or RC link. An optional SD card can be used to support flight logging for offline analysis, debug, and replay in simulation tools.

3.3.2 UAV Integrated Controller

The Unmanned Aerial Vehicle Integrated Controller (UAVIC) merges the FMU and companion computer functionalities into a single platform. This consolidation deploys the PX4 autopilot stack and the video surveillance application into a custom Linux-based OS. However, PX4 was typically designed to run atop the NuttX RTOS, with Linux support remaining limited. Supported platforms include the BeagleBone Blue and Raspberry Pi 2/3/4 with additional shields [105].

Fig. 27 presents the selected UAVIC: the Raspberry Pi 4 with PilotPi shield. This combination was chosen for its detailed documentation and cost efficiency. The PilotPi shield provides a fully functional open-source solution for running PX4 autopilot directly on Raspberry Pi, requiring no proprietary drivers while offering open-source PCB and schematics [106].

[3]Used under the terms of the Creative Commons BY 4.0 license.

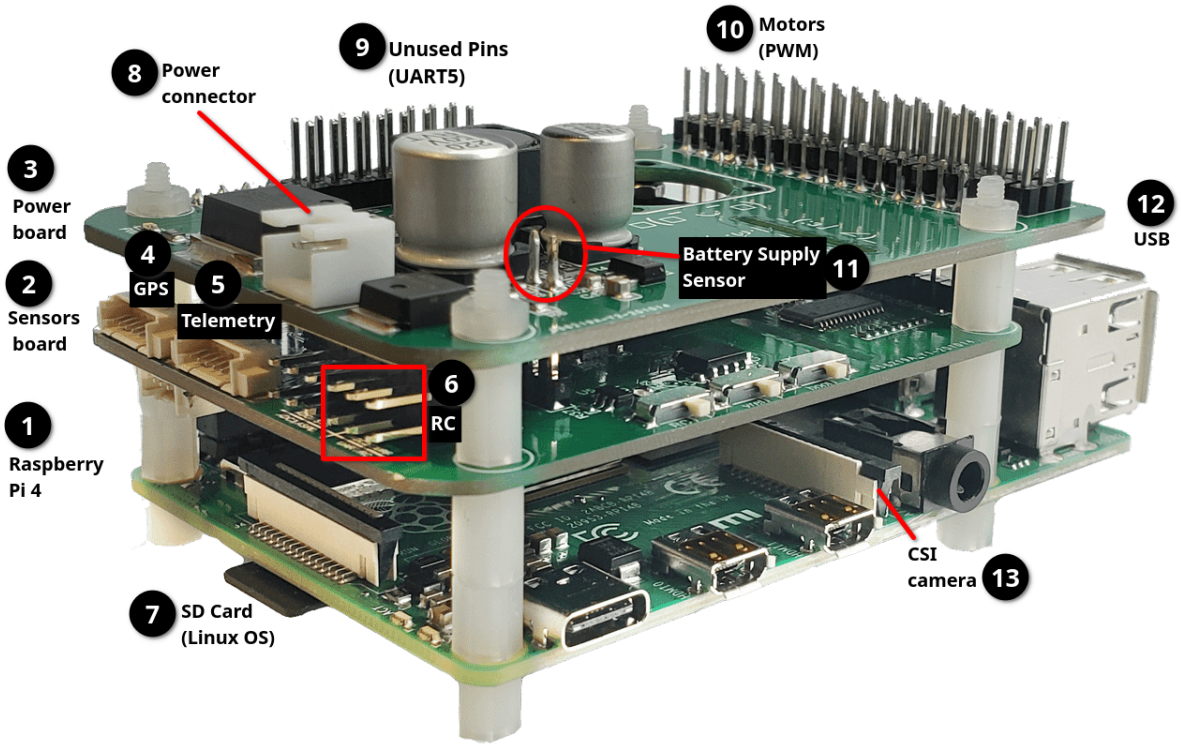


Figure 27: UAVIC: Raspberry Pi 4 + PilotPi shield (adapted from [106][3]
)

The Raspberry Pi 4 Model B (1) operates a Linux-based OS from the SD card, directly exposing the CSI camera and USB interfaces. This model features the Broadcom BCM2711 SoC, containing a 64-bit quad-core Arm®Cortex®–A72 CPU at 1.8 GHz and VideoCore VI GPU at 500 MHz [107, 108], with 8 GB LPDDR4-3200 SDRAM. Connectivity includes dual-band 802.11ac wireless, Bluetooth 5.0, Gigabit Ethernet, two USB 3.0 ports, and two USB 2.0 ports.

The sensor board (2) mounts atop the Raspberry Pi, providing GPS, telemetry, and RC external interfaces mapped to `/dev/ttySC0`, `/dev/ttySC1`, and `/dev/ttyAMA0`, respectively. Onboard sensors include an accelerometer/gyroscope (`ICM42688P`), magnetometer (`IST8310`), and barometer (`MS5611`) required by PX4.

The topmost layer contains the power board (3), handling power supply (8), monitoring (11), and motor actuation via PWM (10) supported by the Linux `PCA9685` driver. A header exposes unused pins, enabling additional connections such as a remote serial interface (UART5).

3.3.3 Hardware mapping

The UAVIC platform must comply with PX4 requirements for available sensors and actuators. Furthermore, the static partitioning nature of the Bao hypervisor necessitates assessing whether the UAVIC hardware is fully available to each guest in the SSPFS solution, or if alternatives are required. This section therefore

proceeds by first mapping the hardware required by PX4 to the Linux device tree for the USPFS solution, followed by adaptation to comply with Bao constraints in the SSPFS implementation.

3.3.3.1 USPFS

Fig. 28 depicts the full device tree for the UAVIC system, representing the USPFS solution (base scenario). The solid lines represent aggregation, i.e., a node that includes another one (e.g., the `root` node includes the `memory` node), and the dashed lines represent dependency, i.e., a node that depends on another one (e.g., the `power` and `firmware` nodes depend on the `mailbox`).

The device tree coloring scheme highlights functional groupings:

- Generic nodes: essential for all Raspberry Pi 4 configurations: `memory`, `cpus`, `power-regulators`, etc.
- PX4-required nodes: `i2c1` (motor actuation), `spi0` (IMU; barometer; magnetometer), `spi1` (GPS; telemetry radio), and UARTs 0/5 (RC link; debug console)
- Companion VM nodes: `i2c0` and `csi1` (camera interface), `mmcnr` (Wi-Fi support)
- Firmware nodes: Enable GPU-CPU communication via `mailbox` [109]

Device tree analysis reveals multiple shared dependencies between PX4 and Companion VM functions: the clock manager (`cprman`), General Purpose Input/Output (GPIO) controller, and Direct Memory Access (DMA) controller. Bao's isolation requirements prohibit device sharing, necessitating architectural alternatives.

The proposed solution maintains PX4's native device requirements while migrating Companion VM devices to the USB interface. As shown by the dependency path, the `usb` device (within `pcie`) shares only the `firmware` node with PX4, which itself depends on the `mailbox`.

While this approach simplifies SSPFS implementation, Bao compatibility remains challenged by the shared `mailbox` requirement. Removal of the `firmware` node from either VM would cause system failure, making this option infeasible.

The adopted approach therefore combines the migration of Companion VM devices to USB and the development of supervised mailbox access management within Bao. This maintains hardware isolation while enabling secure shared resource utilization.

3.3.3.2 Supervised mailbox access

Fig. 29 illustrates the mailbox access for the conventional case (left) and the supervised one (right).

The solid lines indicate synchronous events, while the dashed ones indicate asynchronous events, with the red arrows for the outgoing path as seen by the mailbox driver and the blue ones for the incoming path. The mailbox transactions are represented with white, violet and grey envelopes for the conventional, VM1 and VM2 interactions. The synchronization mechanism (lock) is drawn in brown for the mailbox and in blue for Bao. Hypercalls are represented with a green phone for `Start_TX` hypercall and a red one the `End_TX` hypercall. Lastly, the numbering indicates the sequence of events for both cases.

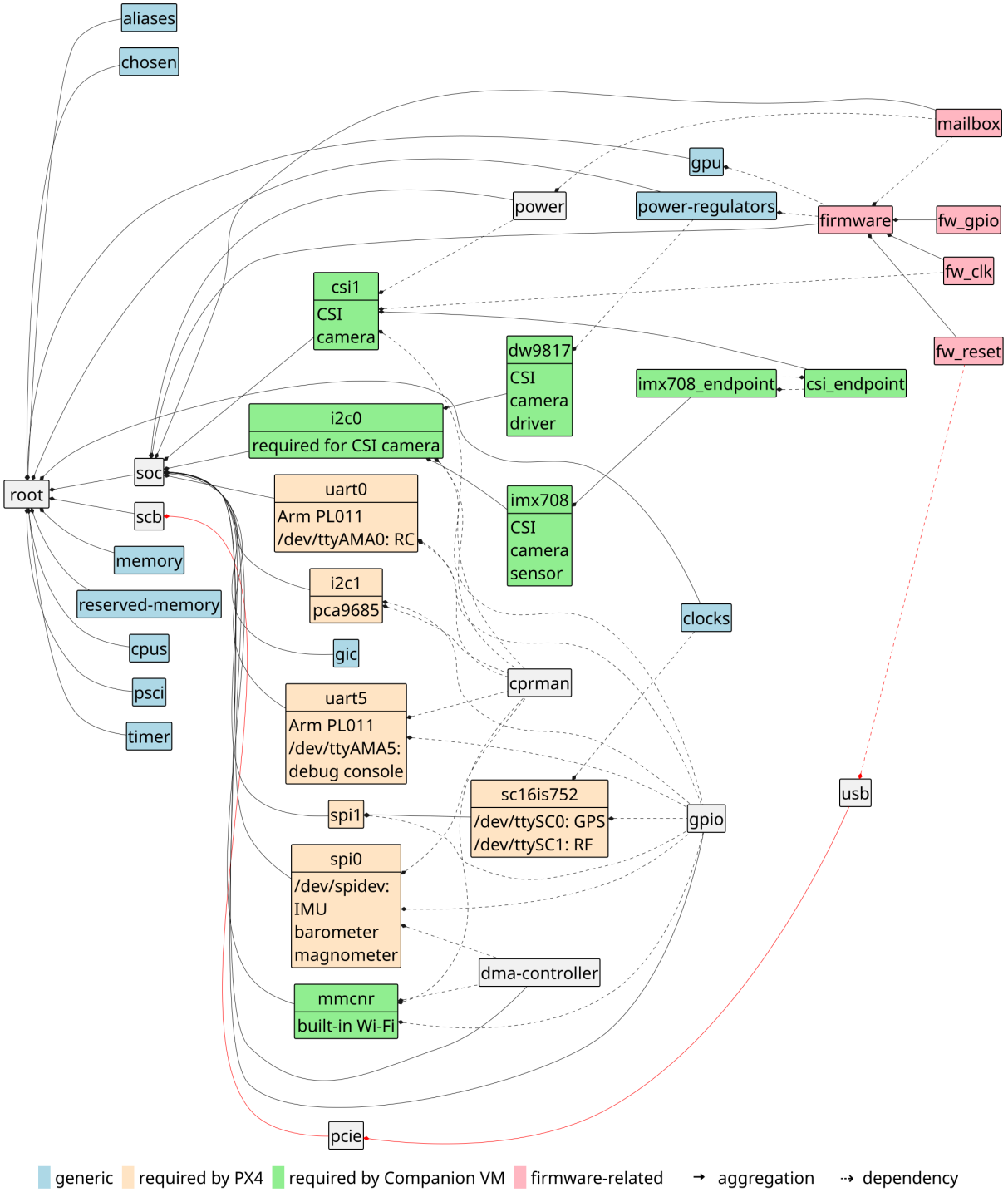


Figure 28: Hardware mapping: USPFS device tree

In the **conventional case**, the mailbox device driver can initiate a transaction request if another transaction is not pending complete (1). The transaction must be completed (8) before a timeout occurs, freeing the mailbox for further requests. An interrupt is triggered (2) and the CPU forwards the request to the mailbox (3) which requests data from the GPU (4). The GPU's firmware handles the transaction and replies back to the mailbox with the result (5). The mailbox forwards the response to the mailbox (6, 7)

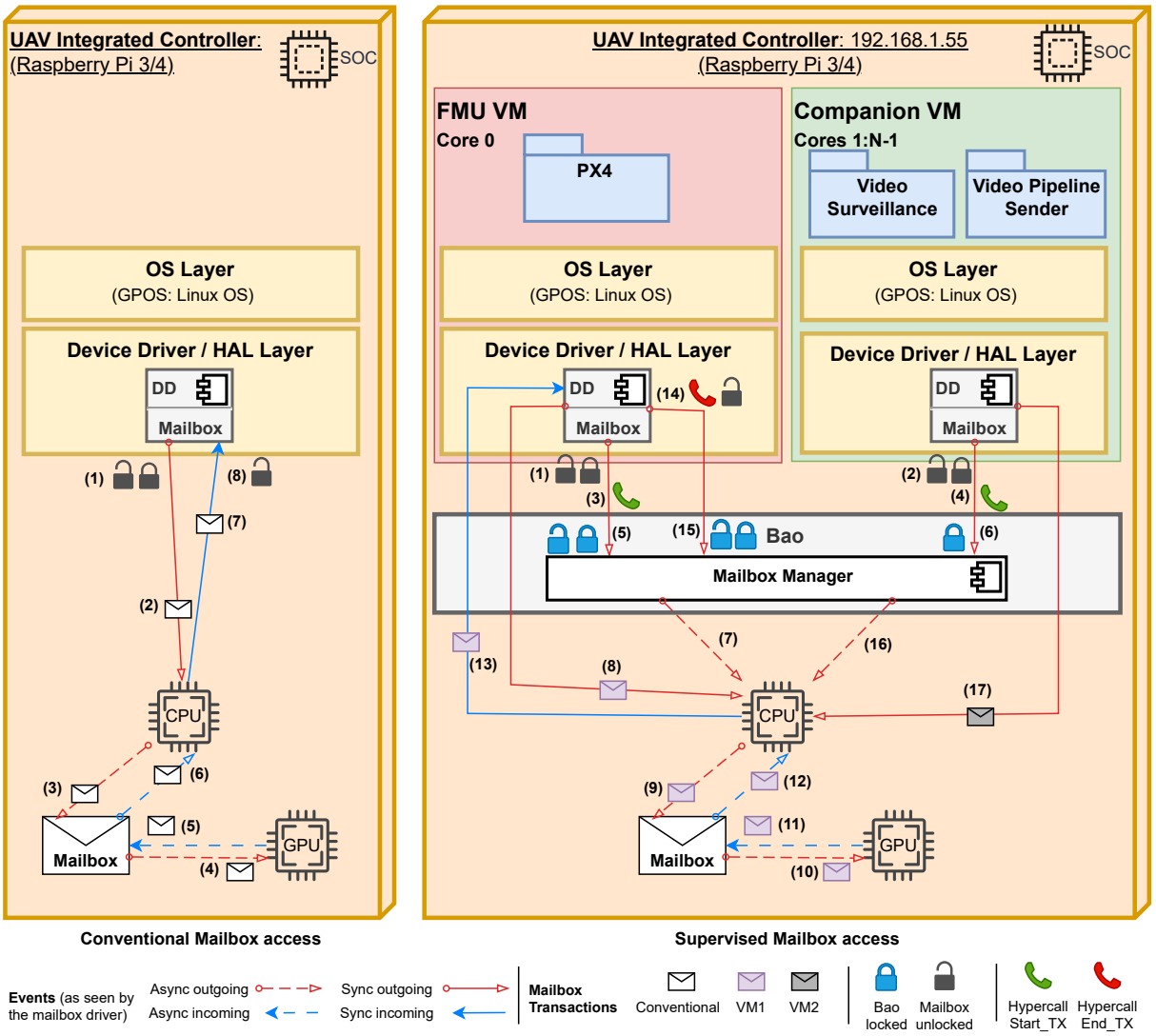


Figure 29: Mailbox access: conventional (left); supervised (right)

completing the request and freeing the mailbox [110].

In the **supervised case**, the VM1 mailbox driver tries to initiate a transaction request if another transaction is not pending complete (1). Now, before sending the transaction, the mailbox driver must signal to Bao it wants to start it by performing a `Start_TX` hypercall (3). Bao acknowledges this request if another one is not pending complete, locking the mailbox manager (5). VM2 tries to do the same (2, 4), but, as VM1 acquired the lock, it must wait for VM1 transaction completion. The Mailbox Manager handles the hypercall, identifying the guest, the target device address (mailbox), and the interrupt ID, and injecting the interrupt in the appropriate CPU (7). The mailbox device driver can now send the transaction to the mailbox (8, 9) which will dispatch it to the GPU (10). The GPU processes the transaction and the response it sent back to the device mailbox driver (11, 12, 13). When the transaction is completed the mailbox driver issues another hypercall — `End_TX` — to signal Bao this event, and the mailbox lock is released, freeing the VM1's mailbox for further requests. Bao handles the hypercall `End_TX` by releasing the mailbox manager's lock (15), which triggers the pending transaction request issued by VM2 (6) to be

resumed by injecting the interrupt in the appropriate CPU (16). Then, the VM2 mailbox's driver can send the firmware transaction and the process continues.

3.3.3.3 SSPFS

After addressing shared devices between VMs through supervised mailbox management, hardware assignment proceeds as follows.

Fig. 30 and Fig. 31 illustrate the device trees for the PX4 VM and Companion VM, respectively.

The PX4 VM configuration incorporates only essential onboard sensors/actuators and an optional debug console. Memory allocation consists of two RAM regions: 144 MB and 3 GB. Within the first region, a 32 MB Contiguous Memory Allocation (CMA) region is reserved to support DMA transactions for SPI devices, which operate exclusively within the first GB of memory [111]. Processing resources include a single Arm A72 CPU (core 0).

The Companion VM configuration incorporates the USB interface for connecting a USB camera and Wi-Fi dongle. Memory allocation features two RAM regions: 624 MB and 2 GB. Within the first region, a 384 MB CMA region supports video pipeline operations, constrained to the first GB of memory [111]. Processing resources include three Arm A72 CPUs (cores 1–3).

3.3.4 Addons

The Companion VM requires two USB devices: a camera and Wi-Fi dongle. Fig. 32 presents the selected USB camera – the Creative Live! Cam Sync 1080p V2 [112].

It is an affordable USB camera with full High-Definition (HD) video capture (1920x1080 @ 30 Frames Per Second (FPS)) and dual built-in microphone. This camera is a plug-and-play device with built-in drivers for Linux available out of the box. Lastly, due to its long cable it can be positioned freely in the UAV.

Fig. 33 presents the selected USB Wi-Fi dongle — the EDUP AX3000. This dual high-gain antena device has tri-band support — 2.4 GHz, 5, and 6 GHz — and it can achieve data transfer rates of up to 3000 Megabits per second (Mbps) on the USB 3.0 interface [113]. It contains a Mediatek `mt7921au` chipset with a wide OS compatibility, supported in-kernel since Linux kernel 5.18 [114].

3.4 Summary

Conventional flight stacks address mixed-criticality through multi-platform designs, increasing UAV weight and footprint. Platform integration represents a key design objective, but requires appropriate supervision to be viable.

This chapter presented three solutions for PX4-based video surveillance applications: conventional (unsupervised multi-platform), unsupervised single-platform (USPFS), and supervised single-platform (SSPFS). The conventional approach established the foundation for platform integration. Subsequent consolidation

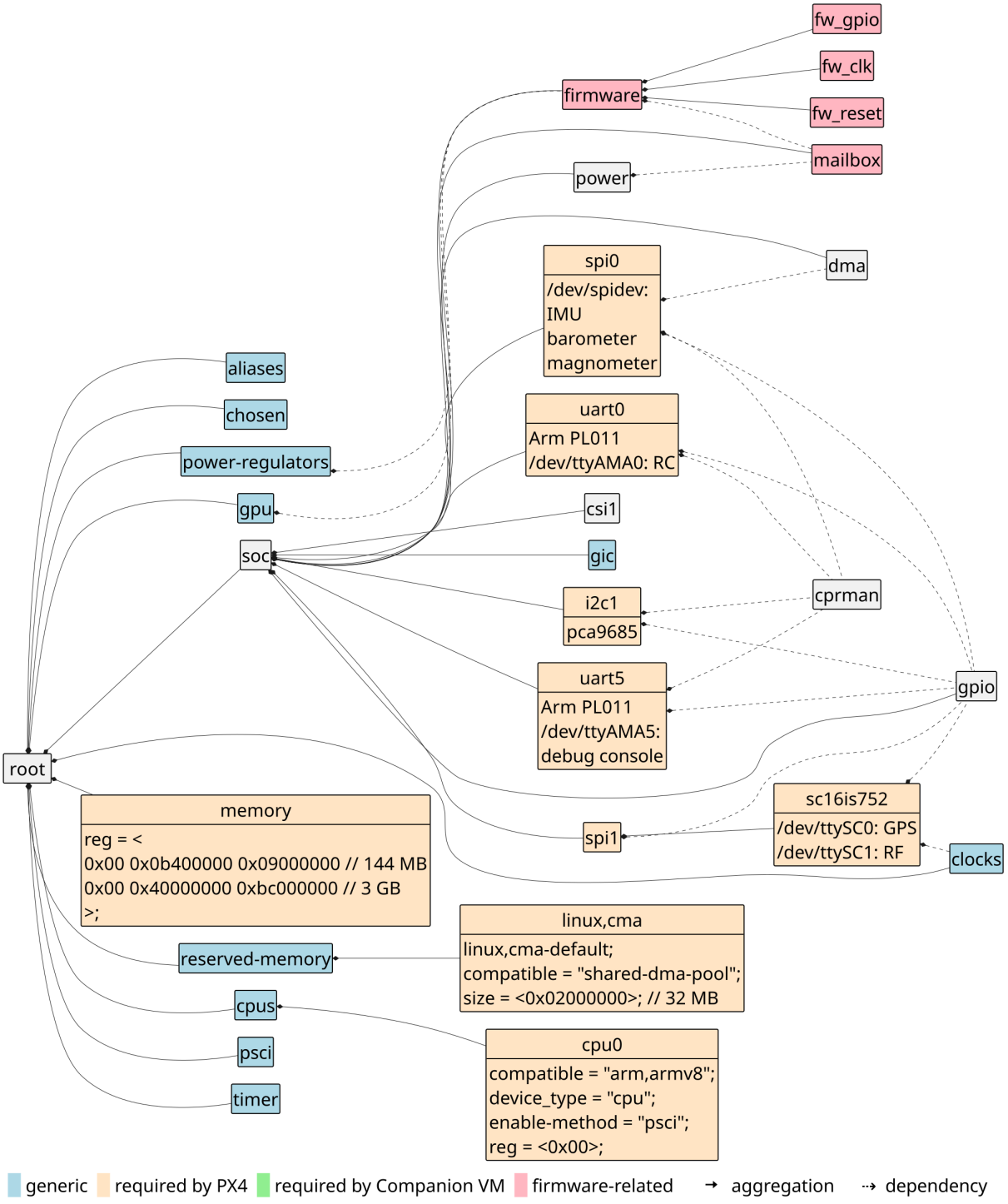


Figure 30: Hardware mapping: SSPFS device tree — PX4

merged flight controller and companion computer functions into the USPFS architecture using a Raspberry Pi 4 with Pilot Pi shield. However, the USPFS provides no isolation between systems, allowing failures in non-critical components to propagate to the FMU. Platform consolidation with system isolation was therefore achieved through Bao hypervisor supervision, forming the SSPFS solution.

Bao's static partitioning requirement creates potential hardware resource conflicts between VMs. This

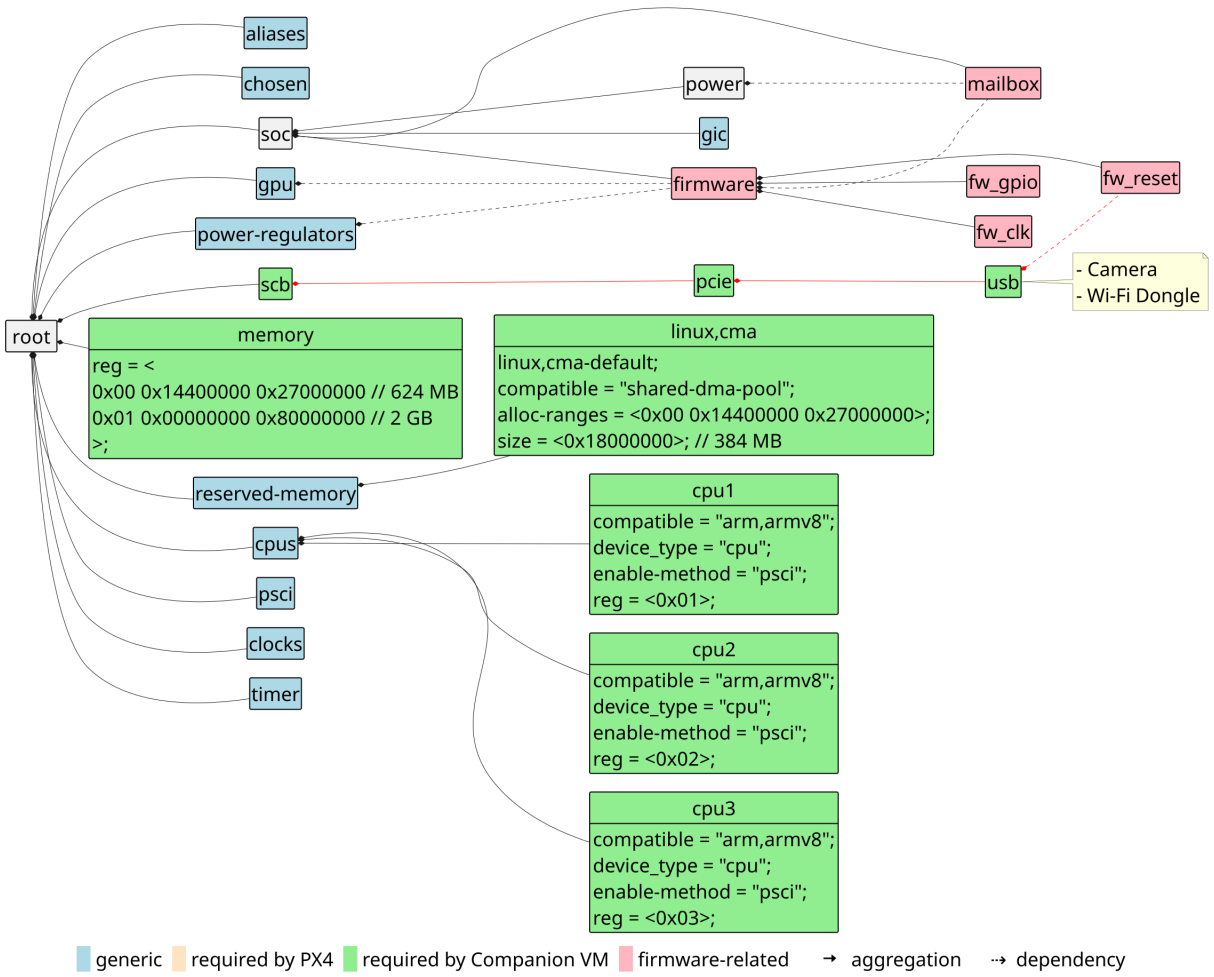


Figure 31: Hardware mapping: SSPFS device tree — Companion VM

Figure 32: Addons: USB Creative Camera

challenge was addressed through UAVIC hardware selection and mapping, producing solution-specific device trees. Identified conflicts were resolved by assigning USB devices (camera and Wi-Fi dongle) to the

Figure 33: Addons: USB Wi-Fi dongle — EDUP AX3000

companion computer VM. One critical shared dependency remained—the firmware mailbox required for Arm CPU-GPU communication. A mailbox access supervision mechanism was consequently developed for Bao, enabling both VMs to communicate securely with the board's firmware.

4

Implementation

"Talk is cheap. Show me the code."
– Linus Torvalds, software engineer

This chapter details the implementation of both USPFS and SSPFS solutions. The implementation workflow begins with an overview of the common methodology. Next, the base system – comprising hardware and software components shared by both solutions – is presented, including initial validation of UAV assembly, configuration, and the PX4/video surveillance stacks. Implementation of the USPFS solution then follows, deploying both software stacks to a custom embedded Linux-based OS. Finally, the SSPFS solution implementation is detailed, featuring deployment of each software stack to separate VMs managed by the Bao hypervisor.

4.1 Workflow

Fig. 34 illustrates the overall implementation workflow. The USPFS solution comprises the **Guests**, **Firmware**, and **Deployment** sections, while the SSPFS solution additionally includes the **Hypervisor** section. The workflow consists of four primary implementation stages: **Build guests**, **Build Hypervisor and VMs** (SSPFS only), **Build Firmware**, and **Deployment**. The term "guest" refers to either a true virtual machine under Bao hypervisor (SSPFS) or a native binary on the UAVIC platform (USPFS).

4.1.1 Guest Construction

The initial stage builds guest components. In the USPFS scenario, a single binary encompasses both PX4 and video surveillance functionality. The SSPFS scenario requires separate PX4 and video surveillance guests executing concurrently with isolation. The PX4 application build process targets the `PilotPi` board's `aarch64` architecture, producing deployable binaries and configuration files. These files are transferred to Root Filesystem Overlays directories alongside network configurations for deployment by Buildroot during guest compilation.

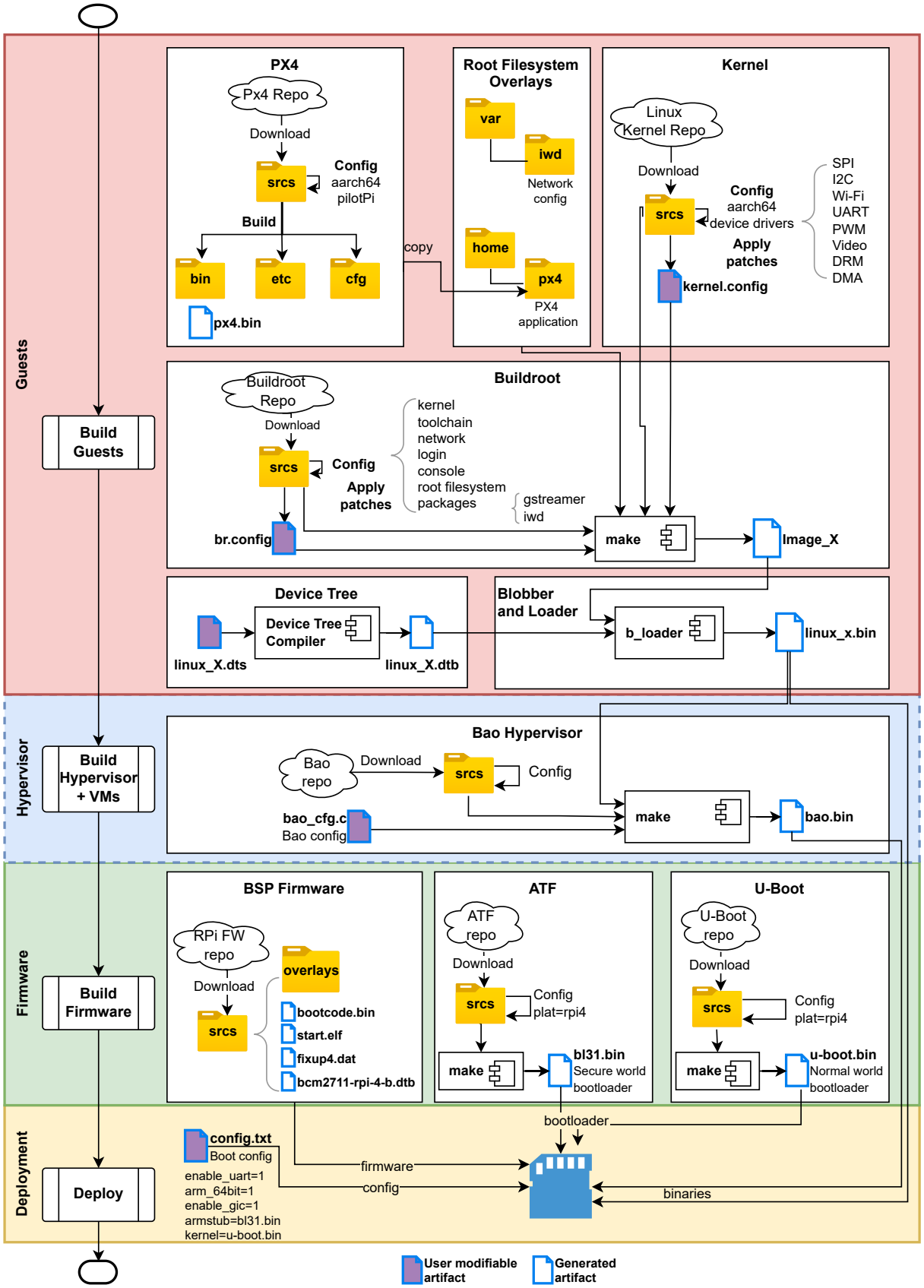


Figure 34: Implementation workflow

The Linux kernel configuration adds support for essential device drivers (SPI, I2C, Wi-Fi, video, etc.). The embedded system configuration incorporates the preconfigured kernel and deploys the PX4 application with network, login, console, and package support (e.g., `gstreamer` for video, `iwd` for wireless). Buildroot compiles a Linux image `Image_X` (where X denotes guest numbering). Device tree source `linux_X.dts` is configured to match guest-specific hardware and compiled to Device Tree Blob (DTB) `linux_X.dtb`. The `b_loader` component combines DTB and Linux image into a deployable executable with minimal dependencies. The USPFS yields one binary; SSPFS requires repetition for each guest, producing two binaries later merged by Bao.

4.1.2 Hypervisor Configuration (SSPFS Only)

The SSPFS solution requires Bao hypervisor configuration via `bao_cfg.c`. This file specifies guest build paths, entry points, CPUs allocation, memory regions, and device memory/interrupt mappings. Configuration data generates a consolidated blob (`bao.bin`) encapsulating both guests atop the hypervisor.

4.1.3 Firmware Compilation

Platform firmware compilation begins with downloading the Raspberry Pi 4 Board Support Package (BSP). The process configures and builds the Arm Trusted Firmware (ATF) (`bl31.bin`), required by Bao, followed by the normal world bootloader (`u-boot.bin`). The bootloader loads the target binary: `linux_x.bin` for USPFS or `bao.bin` for SSPFS.

4.1.4 Deployment

Final deployment writes boot artifacts to the SD card: Raspberry Pi firmware, secondary bootloaders, target binary (`linux_x.bin` or `bao.bin`), and configuration file (`config.txt`). The configuration file initializes the boot process, enabling UART and GIC subsystems while specifying secondary bootloaders for post-initialization execution.

The UAVIC boot flow (Fig. 35) requires detailed analysis to clarify deployment mechanics. The first stage bootloader (`bootcode.bin`) initializes hardware, loads firmware from the SD card, and parses boot parameters from `config.txt`.

Firmware (`start4.elf`) processes `config.txt`, enabling UART and GIC subsystems while specifying sequential loading of secondary bootloaders `bl31.bin` and `u-boot.bin`. Execution then proceeds to `bl31.bin`, which initializes secure services and prepares handoff to the normal world bootloader `u-boot.bin`.

The `u-boot.bin` bootloader initializes peripherals (including console) using the firmware's DTB and environment variables. This stage enables loading and execution of the target binary through either U-Boot script parameters or direct console intervention.

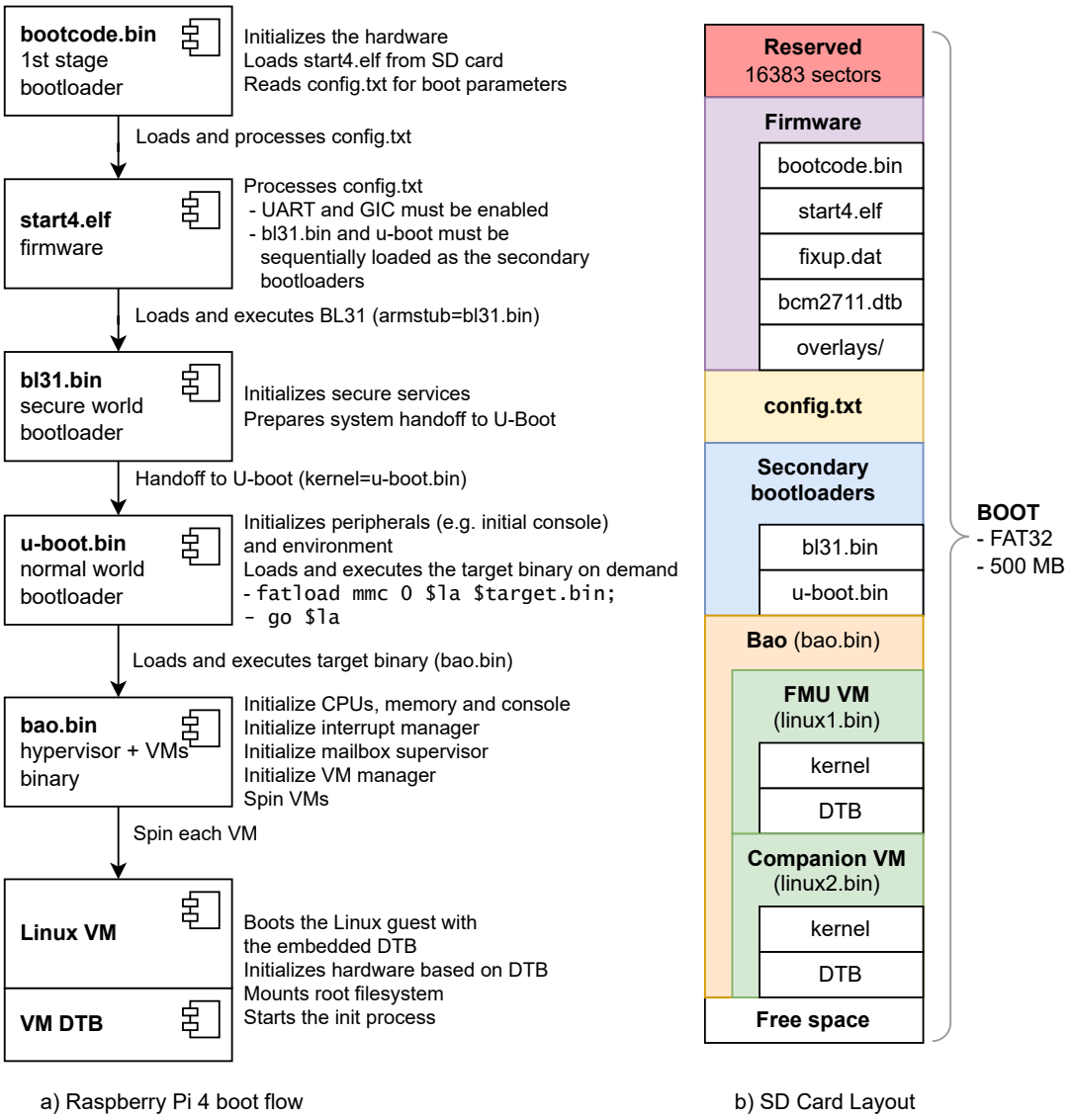


Figure 35: UAVIC boot: a) platform boot flow; b) SD card layout

Target binary execution begins with either `linux_x.bin` (USPFS) or `bao.bin` (SSPFS). Since the latter case encompasses the former, the following analysis focuses specifically on the SSPFS execution path. The Bao hypervisor initialization sequence proceeds through:

1. CPU, memory, and system console initialization
2. Interrupt manager configuration
3. Mailbox supervisor setup
4. VM manager activation

Following initialization, each VM boots its Linux guest using the embedded DTB, initializes hardware per guest configuration, mounts the root filesystem, and launches the `init` process. This results in isolated guest execution under Bao hypervisor supervision.

4.2 Base system

The base system represents the common set of hardware and software components required for the implementation of the USPFS and the SSPFS solutions. It comprises the UAV's assembly and configuration, and the testing and validation of the PX4 and video surveillance stacks.

4.2.1 UAV assembly

The initial implementation phase involves UAV assembly and configuration using the GCS ground station software. Fig. 36 displays the completed UAV based on the HoverGames kit alongside the QGroundControl GCS interface.

Assembly begins with the S500 frame (3) serving as the structural foundation, incorporating landing gear, arms, and electronics support plates. The NEO-M8N GPS module (1) was selected for its cost-effectiveness and urban canyon performance [115]. Four brushless DC motors (2) mount at arm extremities, controlled by optocoupled 40 Ampere ESCs.

Power is supplied by a 3S LiPo battery (12) with 5000 mAh capacity and XT-60 connector (4) [116], providing approximately 30 minutes of full-throttle operation. The UAVIC platform (7) combines Raspberry Pi 4 with PilotPi shield. Telemetry radio links (5)(6) enable communication between the UAV and QGroundControl GCS (11). For development purposes, a debug UART port (8) was included but excluded from final implementation. Video surveillance components – Wi-Fi dongle (9) and camera (10) – connect via USB to the UAVIC.

UAV configuration requires deploying PX4 to the UAVIC before establishing communication with QGroundControl via telemetry radio or Wi-Fi for parameter setup.

4.2.2 PX4

Following assembly validation, PX4 was deployed directly atop a general-purpose Linux OS on the UAVIC platform. Listing B.1 details the PX4 build process: (1) autopilot source download with recursive submodule updates for NuttX applications; (2) Python virtual environment creation and activation; (3) PX4 compilation targeting the PilotPi platform's Arm 64-bit architecture; (4) configuration of autopilot host IP and username for deployment to UAVIC and local backup.

Configuration utilizes the `kconfig` system inherited from NuttX RTOS. Listing C.1 presents a configuration excerpt specifying platform/architecture selection, toolchain setup, and required driver/module enablement.

PX4 initialization employs a boot script for UAV configuration and service activation. Listing C.2 details the `pilotpi_mc.config` startup sequence:

1. Import of saved UAV parameters
2. Vehicle type and airframe configuration
3. Mavlink-triggered camera setup

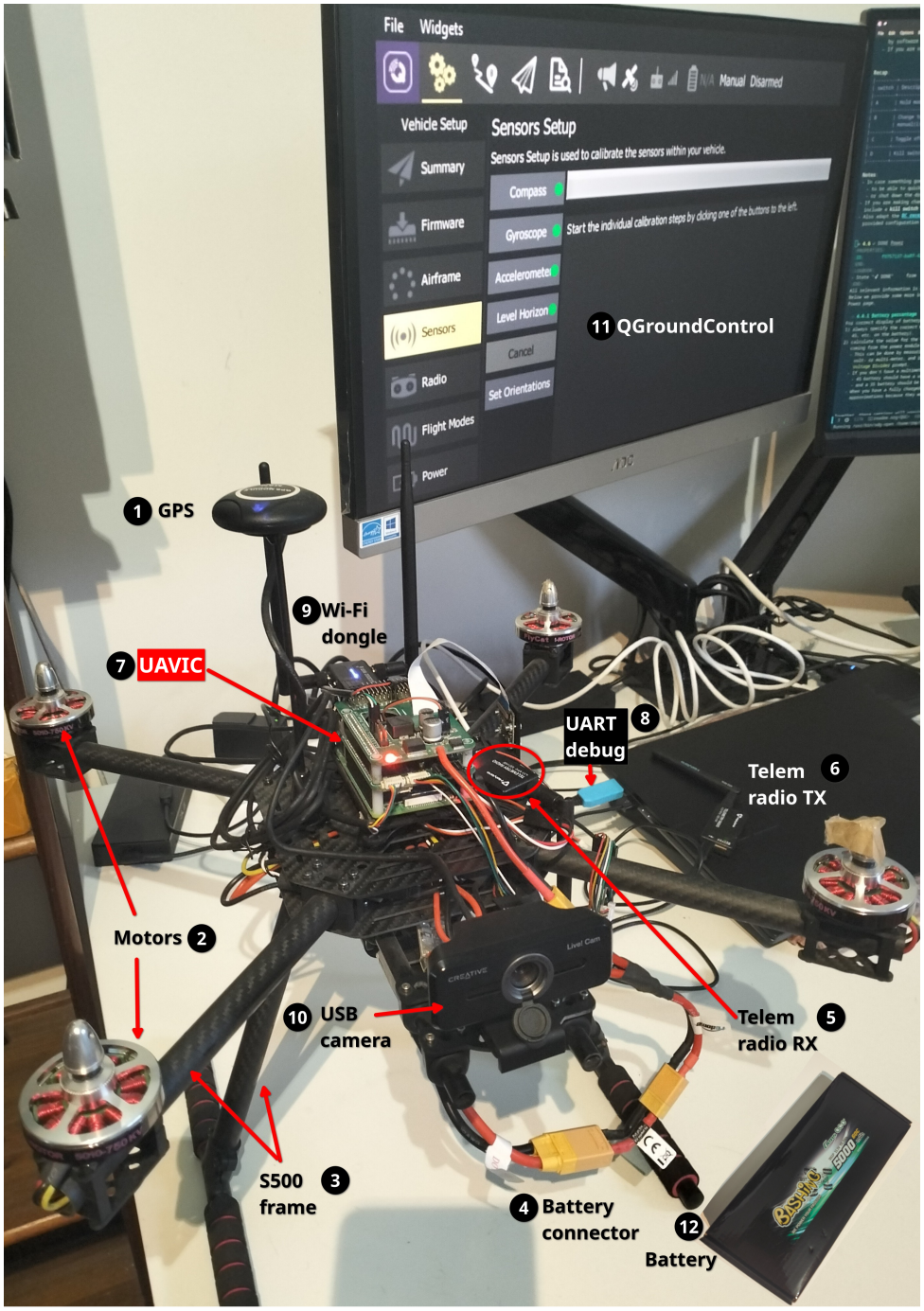


Figure 36: UAV assembly and configuration: UAV (left); Ground Station running QGroundControl (right)

4. Mission/geofence data, CPU monitoring, and battery status service initialization
5. Sensor and actuator initialization
6. RC management task launch
7. Main FMU state machine activation
8. Multicopter task initiation (navigation, position/attitude control, logging)
9. Mavlink communication enablement (telemetry radio and Wi-Fi)

10. Boot completion signaling to GCS for user configuration

Post-deployment execution occurs via `./px4 -s pilotpi_mc.config`. Fig. 37 demonstrates successful PX4 initialization, showing sensor/actuator readiness and active Mavlink communications—specifically the Wi-Fi link enabling GCS connectivity (`partner IP: 192.168.1.37`).

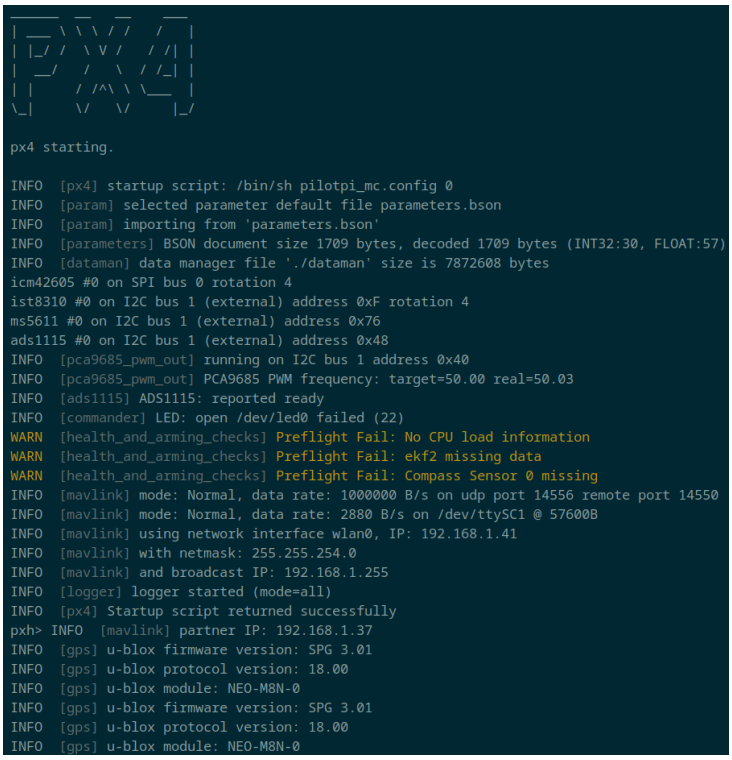


Figure 37: UAV configuration: PX4 boot

4.2.3 UAV configuration

Following UAV-GCS pairing, aircraft configuration proceeds through sequential steps. The initial configuration defines the airframe as a quadrotor X type, specifically the NXP HoverGames variant (Fig. 38).

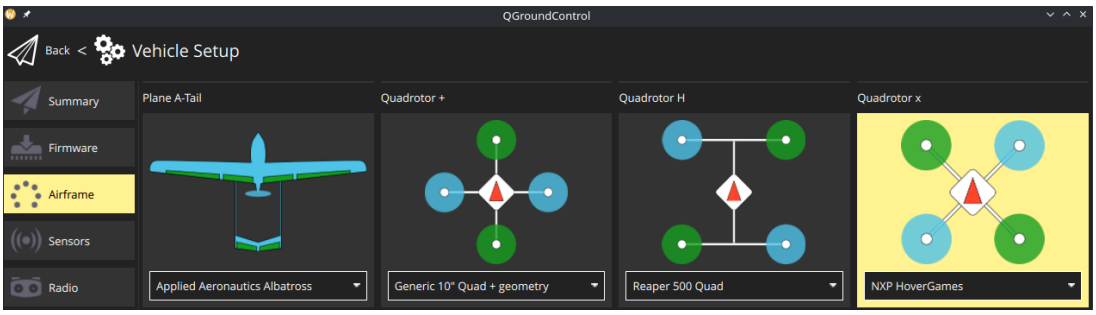


Figure 38: UAV configuration: airframe

Sensor calibration requires physical manipulation of the UAV through prescribed motions to calibrate compass, gyroscope, and accelerometer (Fig. 39).

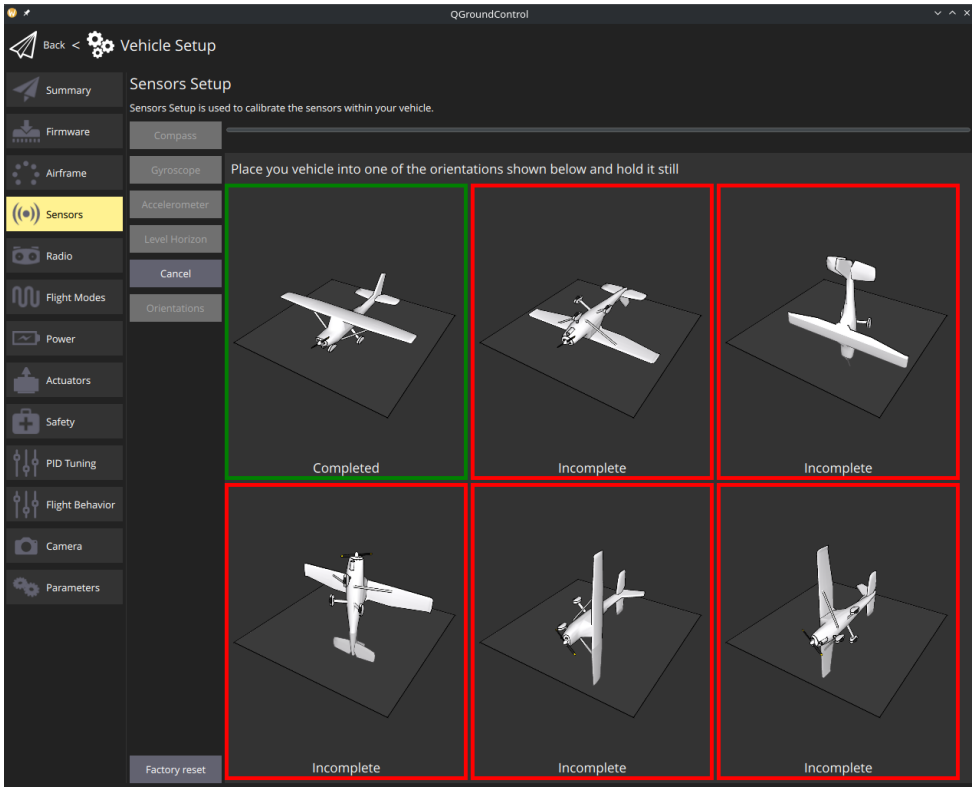


Figure 39: UAV configuration: sensors' calibration

Power management configuration (Fig. 40) involves setting power source parameters, defining battery cell count, calculating voltage divider values, and calibrating ESC PWM minimum/maximum values to prevent output saturation.

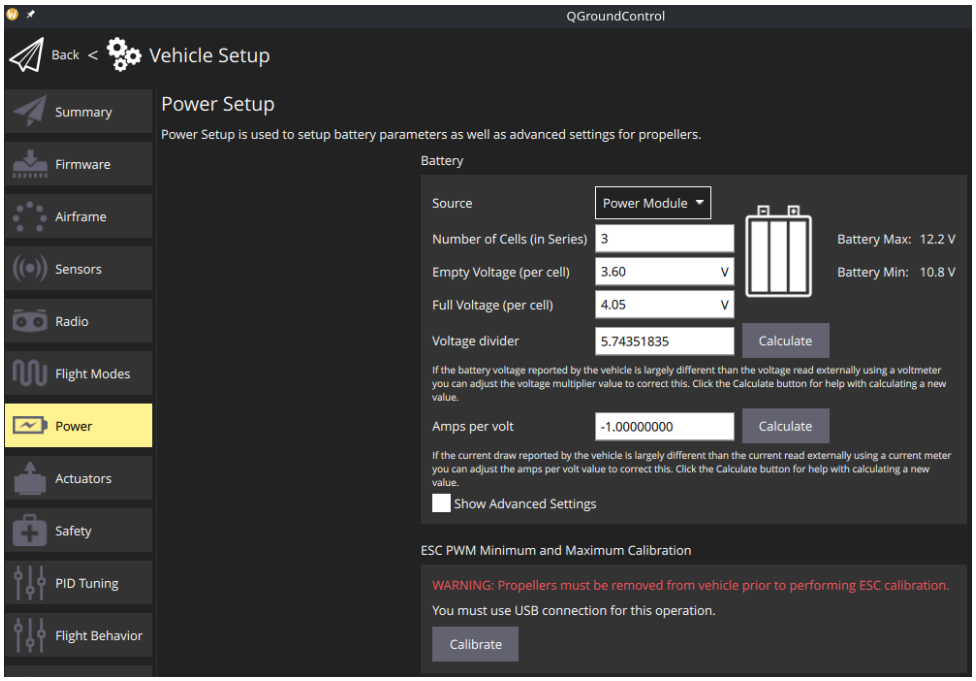


Figure 40: UAV configuration: power management

Actuator calibration (Fig. 41) includes specifying motor count and geometry (position/direction), assigning motors to output channels, and conducting propeller-free motor testing to validate directional rotation alignment with reference diagrams and velocity response to control inputs.

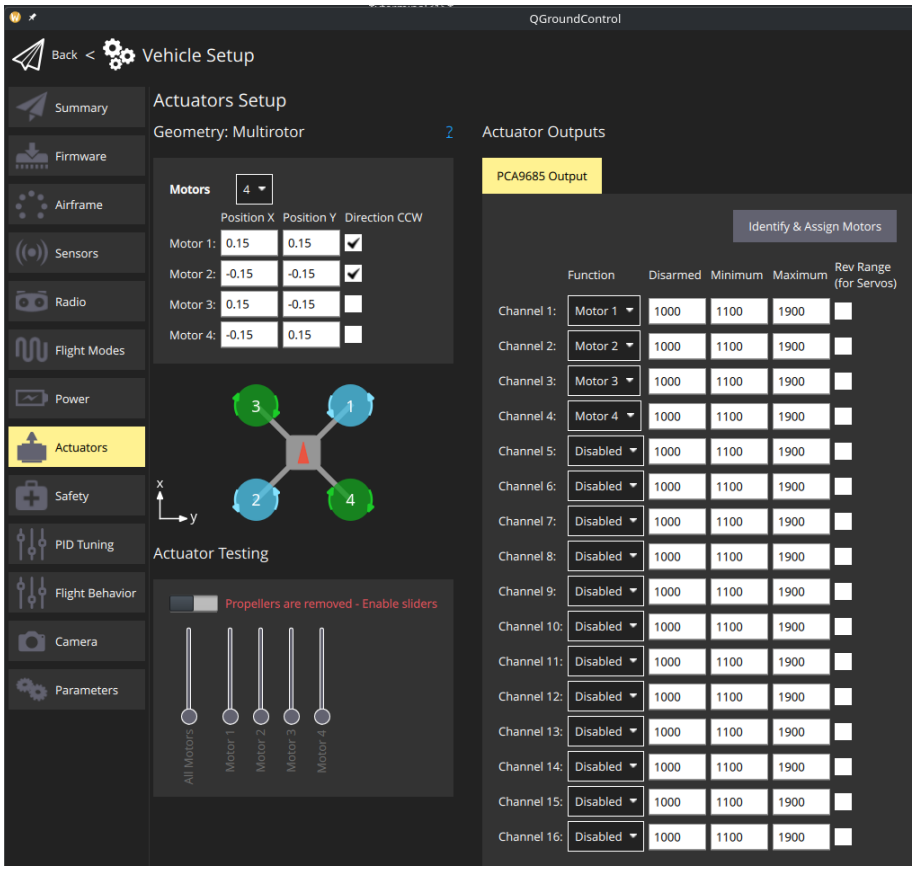


Figure 41: UAV configuration: actuators

Fig. 42 presents the configuration summary, verifying successful parameterization. The established parameter database enables replication across multiple PX4 instances for identical UAV configurations.

4.2.4 Video surveillance

The video surveillance pipeline - designed in Section 3 and illustrated in Fig. 23 - was implemented and validated using the `gstreamer` multimedia framework. This open-source solution was selected for its multi-platform compatibility, modular architecture supporting diverse video codecs, network streaming capabilities, and pipeline configuration flexibility, enabling rapid iteration and simplified testing.

The **sender** component, executing on the UAV, performs frame capture at designated resolution/framerate in MJPEG format, followed by frame decoding, H.264 encoding, and Real-Time Streaming Protocol (RTSP) packaging for UDP transmission.

Listing B.2 details the sender pipeline test script, specifying the USB camera device (`device`) at 640×480 resolution and 30 FPS, along with host IP/port configuration ensuring exclusive GCS reception.

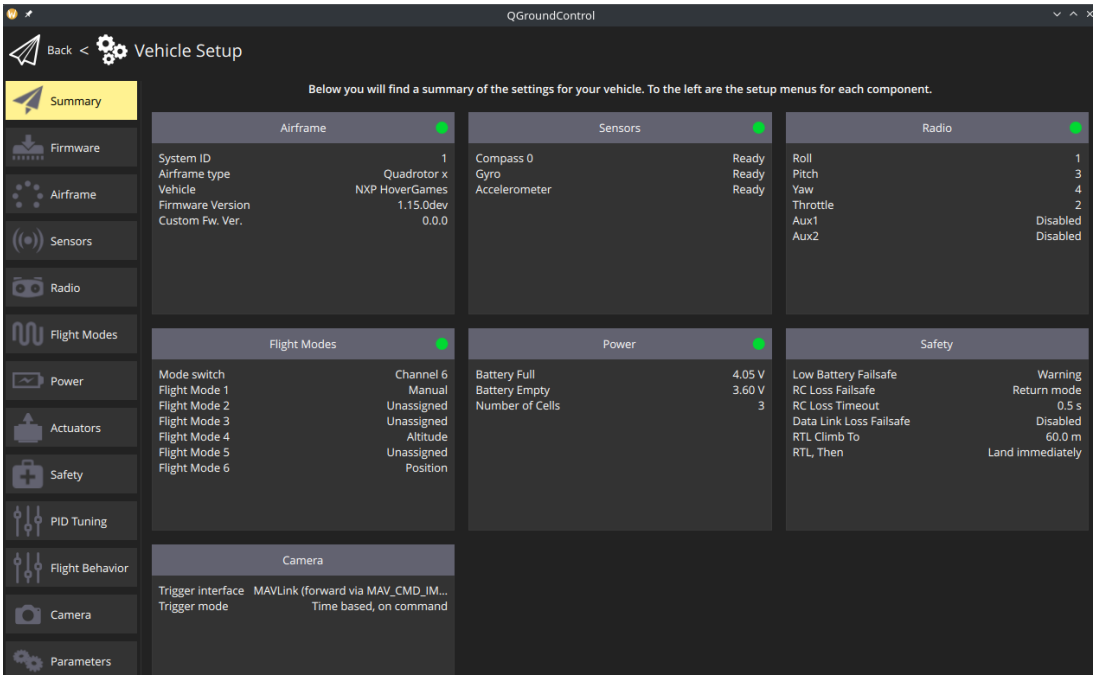


Figure 42: UAV configuration: summary

Conversely, the **receiver** component on the GCS connects to the designated UDP port, decodes and unpacks the RTSP stream, converts to display format, and renders video output.

Listing B.3 shows the receiver test script, configuring the matching UDP port, H.264 decoding, and `autovideosink` for display.

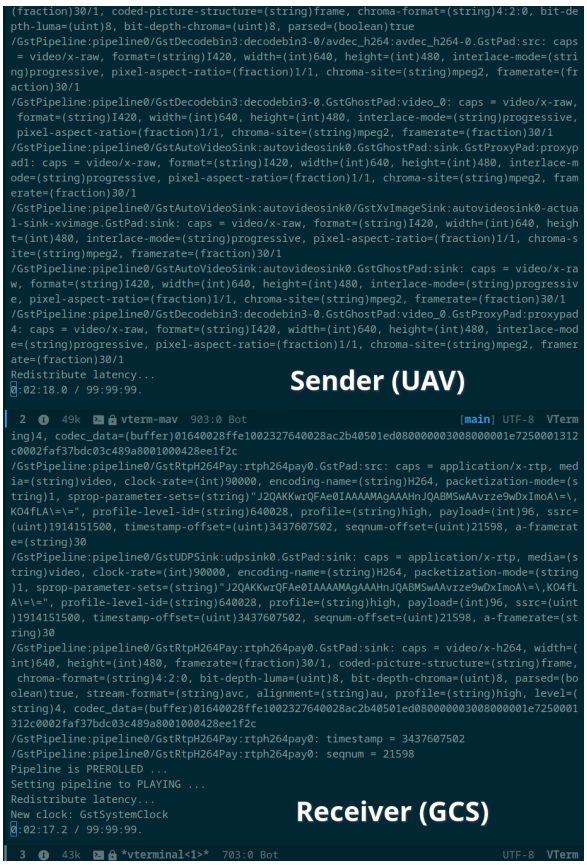
Fig. 43 validates the complete pipeline through a test case demonstrating concurrent execution of the PX4 flight stack (Fig. 43a) and video surveillance sender (Fig. 43b, top) on a generic Linux OS within the UAVIC. Simultaneously, the GCS runs the video receiver (Fig. 43b, bottom) alongside QGroundControl. Fig. 43c confirms successful concurrent operation, displaying UAV telemetry in QGroundControl alongside the video stream.

4.3 USPFS

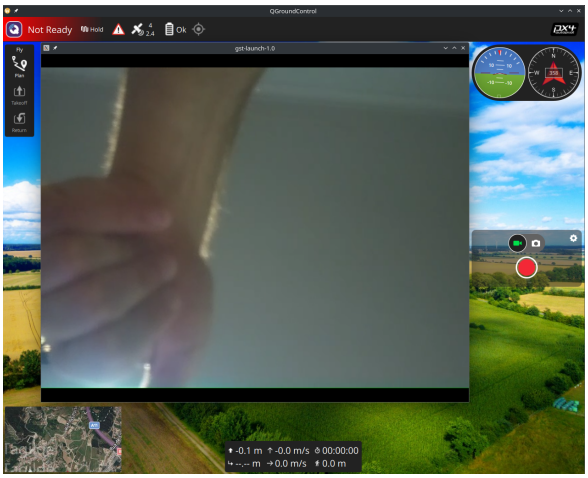
The USPFS solution consolidates PX4 and video surveillance functionality within a custom embedded Linux-based OS on the UAVIC. Kernel configuration requires comprehensive driver support for both software stacks, implemented using the latest stable Raspberry Pi Linux kernel (version 6.6). Listing C.3 details the kernel configuration, including PX4 requirements—SPI/I2C for sensors/actuators, PWM for motors, and PL011 for RC UART—alongside video surveillance components such as video device drivers and the `MT7921U` module for Mediatek Wi-Fi dongle support.

Buildroot configuration (Listing C.4) encompasses multiple critical aspects: preconfigured kernel integration, root filesystem overlay inclusion of PX4 binaries, boot debug console setup to UART5 (`ttyAMA5`), Busybox integration for core utilities, `GStreamer` package addition, and USB Wi-Fi dongle support. This

(a) PX4 boot



(b) Video pipeline: sender (UAV); receiver (GCS)



(c) QGroundControl and video surveillance receiver

Figure 43: Video surveillance pipeline validation

compilation process yields the final OS image `Image_1`.

Device tree customization incorporates all necessary hardware (Fig. 28), with the Linux OS image and device tree combined into executable `linux_1.bin`.

Firmware compilation begins with Raspberry Pi 4 BSP download and secondary bootloader (`bl31.bin`, `u-boot.bin`) configuration. The debug UART5 port (`ttyAMA5`) differs from `bl31.bin`'s default

(UART0), necessitating an ATF source patch (Listing B.4) that replaces `serial0` with `serial5` and modifies the UART offset.

U-Boot configuration (Listing B.5) uses the latest stable release (`v2024.07`) with Raspberry Pi 4 defaults. Environment modification enables automatic binary loading at address `0x80000` following `bl31.bin` execution.

Deployment writes the custom image (`linux_1.bin`), Raspberry Pi firmware, secondary bootloaders, and `config.txt` to the SD card. Listing C.5 shows the configuration file enabling:

- GIC subsystem for early interrupts
- Secondary bootloader specification
- Firmware console setup for early debugging

Platform validation involved:

1. Inserting the SD card into the UAVIC
2. Powering on with UART debug cable connected
3. Monitoring boot process via terminal emulator (`screen /dev/ttyUSB0 115200`)

Listing D.1 displays the boot log, confirming Linux kernel/Buildroot toolchain versions and UART5 console initialization. Final testing applied procedures from Sections 4.2.2 and 4.2.4, verifying correct execution of both PX4 and video surveillance applications in the USPFS environment.

4.4 SSPFS

The SSPFS solution constructs two isolated Linux OSs (guests) running atop the Bao hypervisor: one for PX4 and another for video surveillance. Guest construction parallels the USPFS process but requires duplication. For each guest:

1. Required kernel drivers/packages are selected and compiled into Linux OS images (`Image_X`)
2. Customized Linux Device Tree Source (DTB) files generate executable binaries (`linux_x.bin`) when combined with images

This stage produces two binaries: `linux_1.bin` (PX4) and `linux_2.bin` (video surveillance).

4.4.1 Hypervisor and VM Construction

A Bao hypervisor fork was utilized for platform adaptation. To enable debug console output via UART5 (instead of default UART0), Bao required patching using:

- Platform description file (Listing B.6): Console base address modified to `0xfe201000` (4KB-aligned)
- Platform header (Listing B.7): UART clock set to 48 MHz, address offset to `0xa00` (yielding UART5 address `0xfe201a00`)

Guest resource allocation was defined through custom Bao configurations (Listing C.6):

- VM-image path mapping (lines 1-2)

- Memory region allocation (lines 5-14):
  - PX4: 144MB (first 1GB for DMA) + optional 3GB
  - Video: 624MB (video pipeline) + optional 2GB
  - Additional region for Peripheral Component Interconnect Express (PCIe) bus (USB devices)
- Hardware assignment:
  - PX4 VM: 1 core, 2 memory regions, 9 devices (Fig. 30)
  - Video VM: 3 cores, 2 memory regions, 4 devices (Fig. 31)

Shared devices (architectural timer and mailbox) require special handling: the timer is managed automatically by Bao, while mailbox support enables cross-guest communication.

4.4.2 Mailbox Supervision Implementation

The mailbox supervision mechanism (Fig. 29) integrates two critical components: a mailbox manager addition to Bao (Listing B.8) and Linux kernel mailbox driver patching for Raspberry Pi firmware.

The mailbox manager implements several key features: interrupt definitions and transaction signaling (lines 1-3), `spinlock` synchronization primitive (line 5), interrupt handler injecting interrupts into current vCPU (lines 7-9), and platform initialization registering ISR (lines 12-14). Hypercall callback handling includes transaction start procedures (locking mailbox and enabling interrupt) and transaction end procedures (unlocking mailbox and disabling interrupt).

Bao initialization sequencing (Listing B.9) incorporates Raspberry Pi platform initialization after interrupt management but before VM startup, ensuring proper mailbox registration. Hypercall handling (Listing B.10) now includes mailbox sources (line 1), with transactions triggering the `HC_RPI_FIRMWARE` case (line 15).

The Linux mailbox driver patch (Listing B.11) modifies hypercall source declaration (line 1) and adjusts transaction flow through hypercall insertion before message send (lines 15-21) and after completion (lines 42-48).

4.4.3 Validation

Listings D.2 and D.3 validate mailbox supervision under concurrent guest access through a testing methodology employing dual access channels: PX4 VM via UART5 and Video VM via SSH over Wi-Fi. Successful transaction execution by both guests confirms implementation correctness.

The firmware building procedure mirrored the approach detailed in Section 4.3. Boot artifacts deployed to the SD card included Raspberry Pi firmware, secondary bootloaders (`bl31.bin`, `u-boot.bin`), boot configuration file (`config.txt`), and the Bao hypervisor executable (`bao.bin`) encapsulating both guests. Optional staged validation incorporated guest binaries (`linux_1.bin`, `linux_2.bin`) for individual guest testing prior to integrated hypervisor execution.

SSPFS validation executed `bao.bin` on the UAVIC platform using the command: `la=0x80000; fatload mmc 0 $la bao.bin; go $la`. Guest boot inspection employed separate channels: PX4 VM through UART debug console (`screen /dev/ttyUSB0 115200`, Listing D.4) and Video VM via SSH over Wi-Fi (`ssh root@192.168.1.X`, Listing D.5).

Listing D.4 confirms Bao execution initialization (line 2), Linux OS build details (lines 3-6), memory configuration including CMA/DMA regions (lines 10-22), UART5 console operation, successful mailbox firmware access (lines 24-25), SPI device initialization (lines 34-35), and the Buildroot welcome message (line 49).

Listing D.5 verifies consistent Linux build information (lines 8-10), memory configuration details (lines 15-30), tri-core CPU boot sequence (lines 32-40), validated mailbox supervisor operation (lines 42-43), USB device enumeration for both camera and Wi-Fi dongle (lines 47-56), automated Wi-Fi authentication via `iwd` (lines 58-64), and network configuration confirmation (lines 73-80).

4.4.4 Application Validation

Post-boot verification extended to application execution analysis. PX4 validation utilized QGroundControl with telemetry radio link, where the `work_queue status` command (Fig. 44) confirmed operational status across critical threads including control, navigation, communications, and sensor tasks, thereby validating both telemetry functionality and PX4 execution integrity.

```
pxh> work_queue status

Work Queue: 8  threads                          RATE        INTERVAL
|__ 1) wq:rate_ctrl
|   |__ 1) control_allocator                  400.0 Hz         2500 us
|   |__ 2) mc_rate_control                    400.0 Hz         2500 us
|   \__ 3) vehicle_angular_velocity           400.0 Hz         2500 us
|__ 2) wq:SPI0
|   \__ 1) icm42605                           398.7 Hz         2508 us (2500 us)
|__ 3) wq:I2C1
|   |__ 1) ist8310                             47.1 Hz        21251 us
|   |__ 2) ms5611                              90.8 Hz        11015 us
|   \__ 3) pca9685_pwm_out                     50.2 Hz        19909 us (19949 us)
|__ 4) wq:nav_and_controllers
|   |__ 1) flight_mode_manager                 50.1 Hz        19962 us
|   |__ 2) land_detector                      100.1 Hz         9992 us
|   |__ 3) mc_att_control                     200.1 Hz         4999 us
|   |__ 4) mc_hover_thrust_estimator          100.1 Hz         9992 us
|   |__ 5) mc_pos_control                     100.1 Hz         9992 us
|   |__ 6) sensors                            200.0 Hz         4999 us
|   |__ 7) vehicle_acceleration               200.0 Hz         4999 us
|   |__ 8) vehicle_air_data                    68.2 Hz        14671 us
|   |__ 9) vehicle_gps_position                 3.4 Hz       291756 us
|   \__10) vehicle_magnetometer                47.1 Hz        21252 us
|__ 5) wq:INS0
|   |__ 1) ekf2                               200.0 Hz         4999 us
|   \__ 2) vehicle_imu                        200.0 Hz         4999 us
|__ 6) wq:hp_default
|   |__ 1) battery_status                       0.0 Hz            0 us
|   \__ 2) rc_update                            0.0 Hz            0 us
|__ 7) wq:ttyUnknown
|   \__ 1) rc_input                           250.0 Hz         4000 us (4000 us)
\__ 8) wq:lp_default
    |__ 1) load_mon                             2.0 Hz       489687 us (500000 us)
    \__ 2) send_event                          30.0 Hz        33306 us (33333 us)
```

Figure 44: SSPFS: PX4 system analysis via `work_queue status`

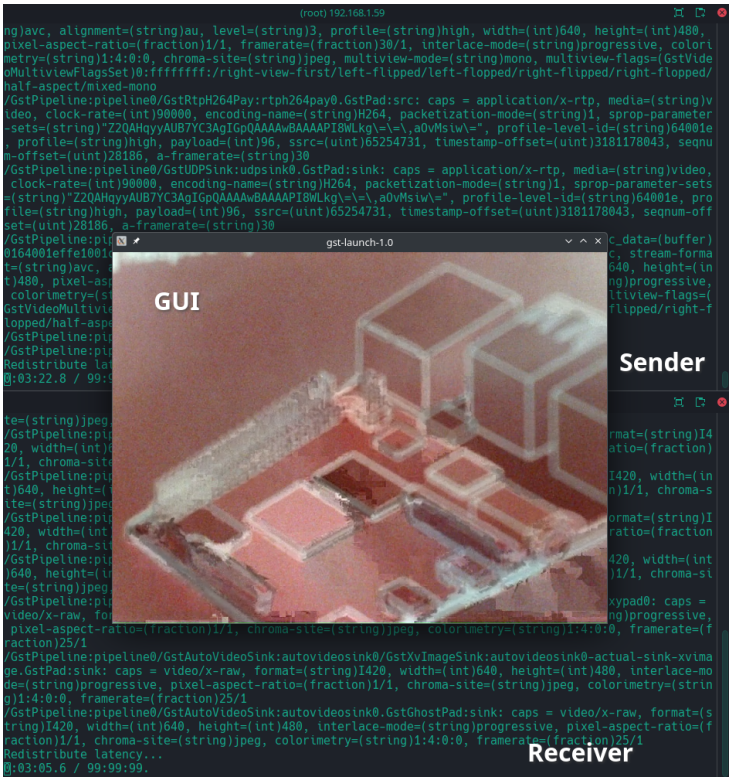


Figure 45: SSPFS: Video surveillance application

Video surveillance validation (Fig. 45) demonstrated full pipeline functionality within the video surveillance VM through coordinated operations: sender application execution on UAVIC (top terminal, `192.168.1.59`), receiver application operation (bottom terminal), and live video streaming display via receiver GUI (middle panel).

4.5 Summary

This chapter detailed the implementation of both USPFS and SSPFS solutions following the established workflow. Base system validation encompassed four critical stages: UAV physical assembly, PX4 compilation and deployment for the target platform, UAV configuration via PX4, and video surveillance pipeline setup with validation.

The USPFS implementation featured deployment of a custom embedded Linux OS consolidating PX4 and video surveillance stacks, supported by kernel configuration for required drivers/modules. Key modifications included ATF patching for UART5 console redirection and boot process configuration through U-Boot environment scripts and `config.txt`, culminating in successful validation of both software stacks.

The SSPFS implementation involved constructing isolated Linux guests for each software stack, with Bao hypervisor customization adding UART5 support. Resource allocation assigned PX4 (1 CPU, 144MB RAM, 9 devices) and Video (3 CPUs, 624MB RAM, 4 devices) their respective resources. Mailbox supervision implementation required coordinated Bao hypervisor modifications and Linux kernel mailbox

driver patching. Validation occurred through boot process inspection via UART debug/SSH and functional verification of both software stacks.

Both solutions were successfully implemented and validated, meeting all functional requirements.

5 Evaluation

"Without data, you're just another person with an opinion."
– W. Edwards Deming, statistician

This chapter presents comprehensive validation of the unsupervised (USPFS) and supervised (SSPFS) flight stack solutions through three complementary assessment domains.

Security Analysis comparatively evaluates system behavior under malicious compromise scenarios, focusing on isolation effectiveness between critical and non-critical components.

Performance Benchmarking conducts quantitative assessment using the MiBench Automotive and Industrial Control Suite (AICS) to establish baseline performance of the USPFS system, quantify hypervisor-induced overhead in SSPFS, measure interference impact between virtualized guests, and evaluate mailbox driver patch consequences. Guest-specific metrics are also evaluated including PX4 task scheduling overhead and camera frame rates.

Operational Validation performs real-flight evaluation via automated missions to compare position tracking accuracy and system resource utilization patterns.

This multi-faceted methodology provides empirical evidence for security guarantees, performance characteristics, and operational viability across both architectural approaches.

5.1 Functional tests

Fig. 46 outlines the functional test methodology, which evaluates system resilience when compromised by malicious actors. Attack scenarios were emulated through deployment of malicious components—either in user space (`crash_app`) or kernel space (`crash_mod.ko`)—to the UAVIC platform in both architectural configurations.

In the **unsupervised** configuration:

1. Malicious components execute directly on the host Linux OS
2. System-wide crash occurs
3. Both non-critical (video surveillance) and critical (PX4) stacks terminate catastrophically

4. Resultant UAV failure ensues

In the **supervised** configuration:

1. Malicious components deploy exclusively to the Companion VM
2. Hypervisor fault triggers VM crash
3. Non-critical stack terminates while critical PX4 stack remains operational
4. UAV maintains flight capability

This comparative analysis demonstrates the supervised architecture's ability to contain faults within non-critical domains, preserving essential flight functions during security compromises.

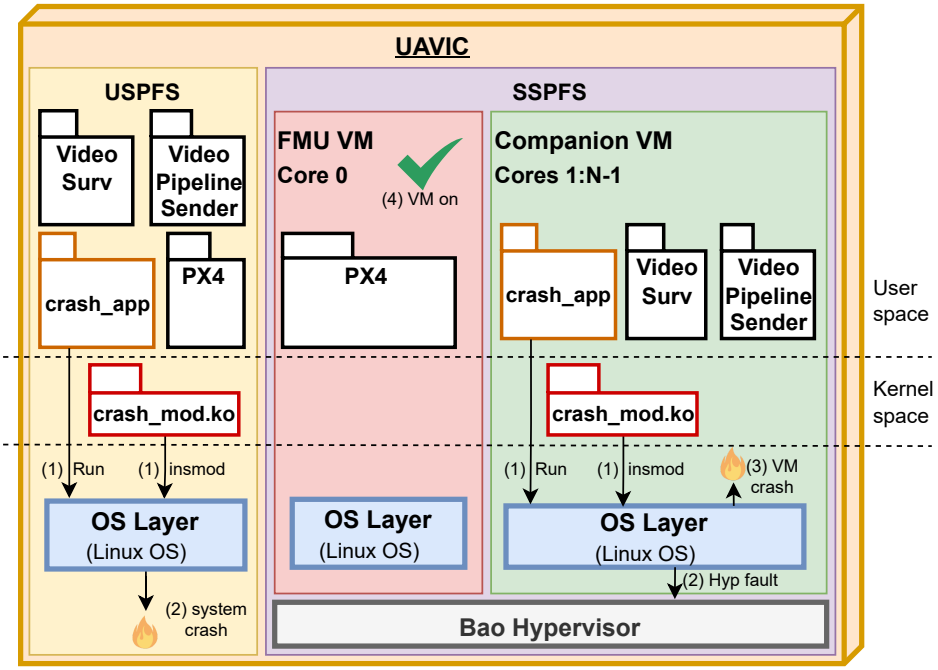


Figure 46: Functional tests' overview

5.1.1 Privileged mode

Malicious actors gaining privileged system access – superuser or kernel space execution – pose significant threats. This section examines system vulnerability to privileged-mode attacks through malicious kernel module deployment.

5.1.1.1 Kernel Panic Attack

A fundamental kernel attack vector involves explicit `panic` invocation, signaling unrecoverable system states. Listing B.12 (line 5) implements this approach. Execution triggers [117]:

1. Local interrupt and preemption disabling
2. "Panic master CPU" identification with forced halting of other CPUs
3. `Forced kernel panic!` message logging (inspectable via `dmesg`)

4. System reboot after timeout expiration, or
5. Infinite loop freezing system operation

Fig. 47 demonstrates panic module testing in the USPFS system:

- System boot via UART (top left, 1)
- Remote kernel module insertion via SSH over Wi-Fi (top right, 2)
- Subsequent network unreachability (bottom, 3) confirming catastrophic UAV failure

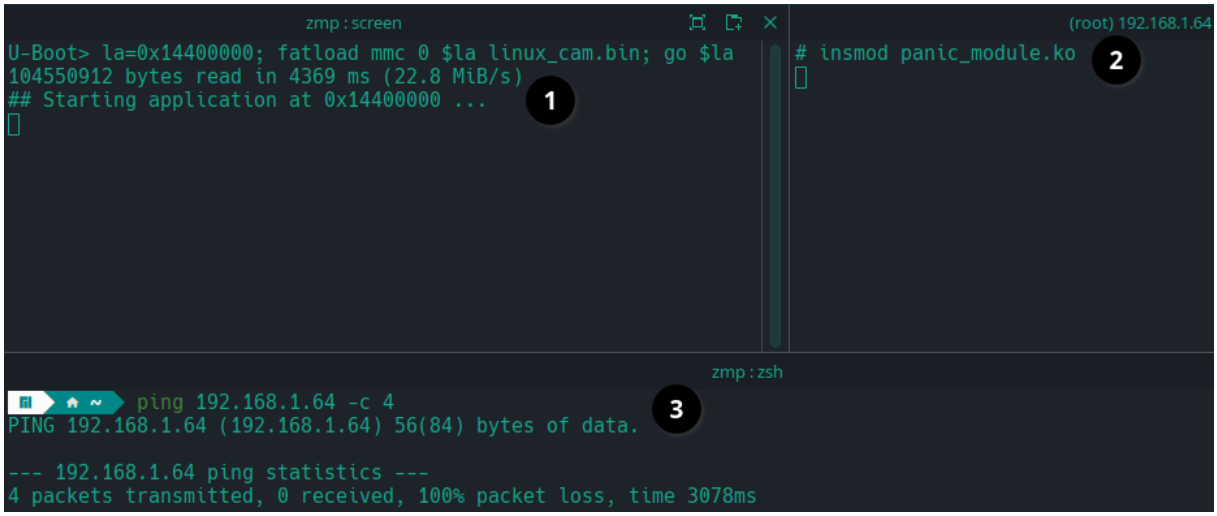


Figure 47: Functional tests: Panic kernel module testing – USPFS

Fig. 48 shows equivalent testing in the SSPFS system:

- Shared UART boot console (Bao/PX4 VM, top left, 1)
- Malicious module insertion in Companion VM (SSH over Wi-Fi, top right, 2)
- Companion VM network failure (bottom, 3)
- Maintained PX4 VM operation (4) confirming UAV operational integrity

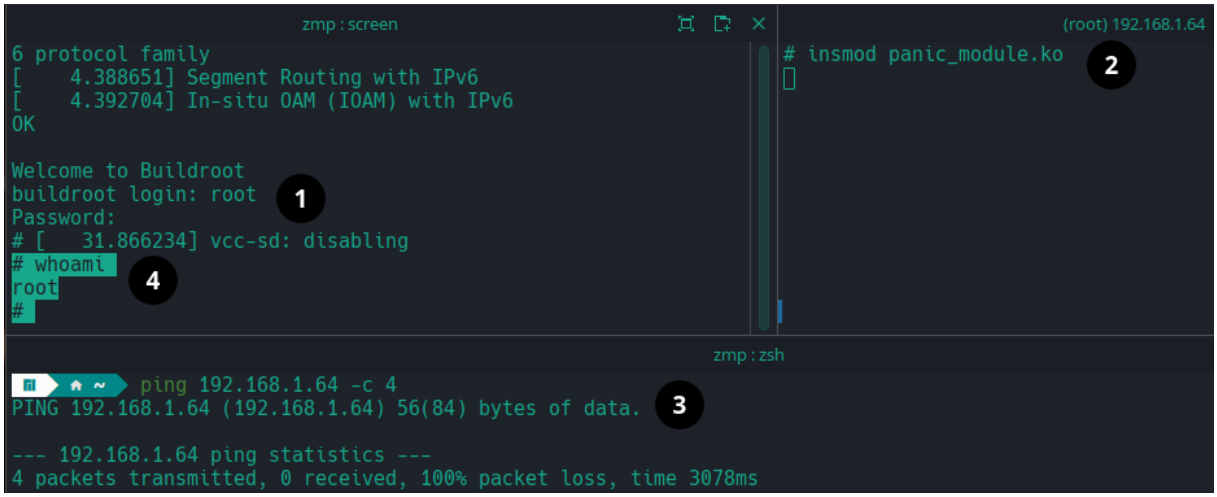


Figure 48: Functional tests: Panic kernel module testing – SSPFS

These results demonstrate Bao hypervisor's capacity to consolidate mixed-criticality stacks while maintaining domain isolation during privileged-mode attacks.

5.1.1.2 Memory corruption

Memory corruption attacks exploit writes to unmapped physical addresses as a system compromise vector. Listing B.13 implements this approach with parameterizable target addresses (lines 9-10). The module:

1. Maps a valid but non-existent physical address to kernel virtual space (line 27)
2. Writes data to this address (line 36)
3. Triggers a crash due to lack of physical backing [118]

Fig. 49 demonstrates memory corruption testing in the USPFS system:

- System boot via UART (top left, 1)
- Remote module insertion via SSH over Wi-Fi (top right, 2) using invalid address `0x8_0000_0000` (beyond Raspberry Pi 4's `0x7_FFFF_FFFF` range [108])
- Resultant network unreachability (bottom, 3) confirming catastrophic UAV failure

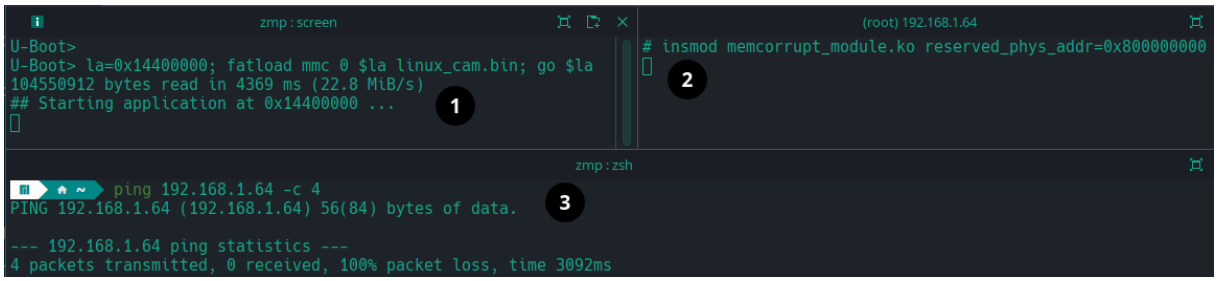


Figure 49: Functional tests: Memory corruption kernel module testing – USPFS

Fig. 50 shows equivalent testing in the SSPFS system:

- Shared UART boot console (Bao/PX4 VM, top left, 1)
- Malicious module insertion in Companion VM (SSH over Wi-Fi, top right, 2) using `0x8_0000_0000`
- Hypervisor fault triggered at invalid virtual address (3)
- Companion VM network failure (bottom, 4)
- Maintained PX4 VM operation (5) preserving UAV functionality

5.1.1.3 Memory exhaustion

Memory exhaustion attacks represent another viable system compromise vector (Listing B.14). The attack methodology involves:

1. Adjusting the Out Of Memory (OOM) score of the executing kernel thread to minimum priority (line 9)
2. Reducing likelihood of termination by the OOM killer during memory pressure
3. Entering an infinite loop (line 11) that repeatedly allocates single memory pages and locks pages (line 16) to prevent reclamation

Figure 50: Functional tests: testing of the memory corruption kernel module – SSPFS

This permanently "hogs" memory resources, preventing reclamation even under severe memory constraints.

Fig. 51 and Fig. 52 demonstrate memory exhaustion testing in the USPFS and SSPFS systems respectively. Consistent with previous attack vectors, results show:

- Catastrophic system failure in the unsupervised (USPFS) configuration
- Contained failure limited to the Companion VM in the supervised (SSPFS) architecture

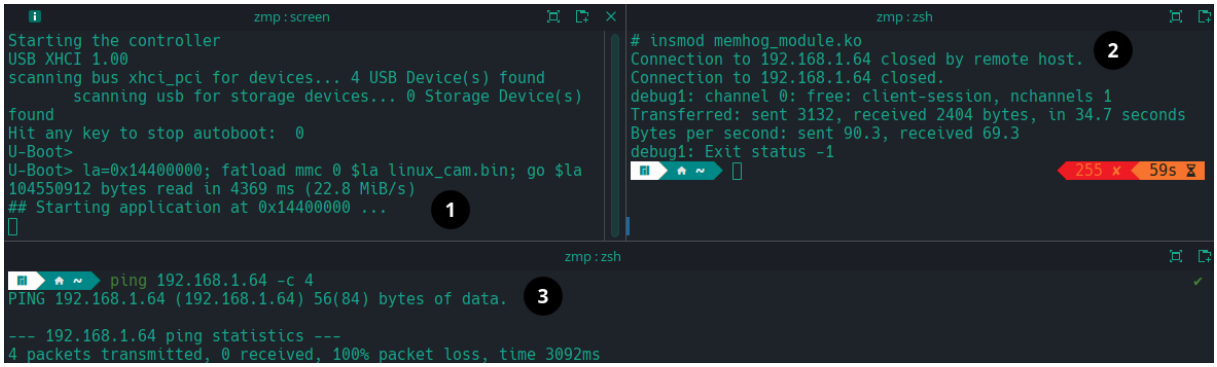


Figure 51: Functional tests: testing of the memory exhaustion kernel module – USPFS

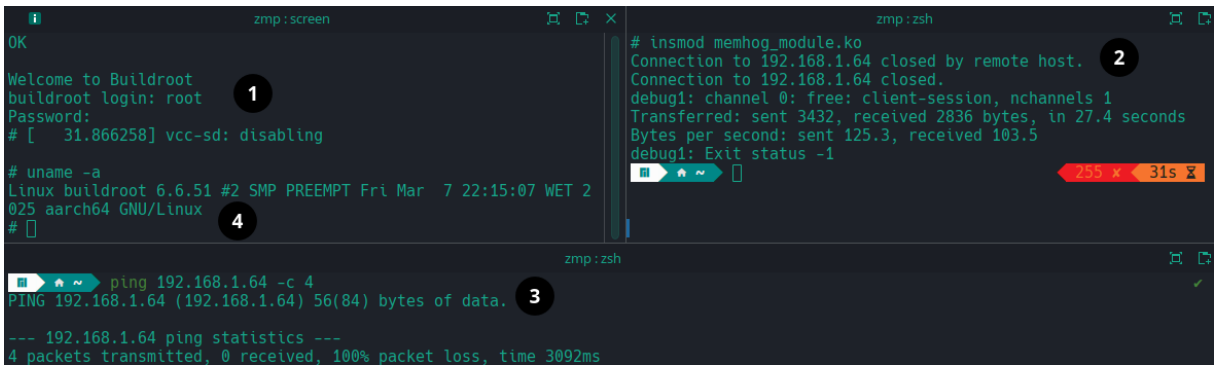


Figure 52: Functional tests: testing of the memory exhaustion kernel module – SSPFS

5.1.1.4 Init process termination

Terminating the `init` process provides a final attack vector for system compromise (Listing B.15). The methodology involves:

1. Acquiring a reference to the `init` process (line 16)
2. Retrieving its memory layout (line 19)
3. Iterating through all Virtual Memory Areas (VMAs) (line 25)
4. For each page (line 27):
    - Pinning to prevent swapping (line 32)
    - Data corruption (line 38)

This comprehensive corruption of the `init` process's memory space induces system failure.

Fig. 53 and Fig. 54 demonstrate init process termination testing in the USPFS and SSPFS systems respectively. Results align with previous attack patterns:

- Catastrophic failure in the unsupervised (USPFS) configuration
- Contained failure limited to the Companion VM in the supervised (SSPFS) architecture

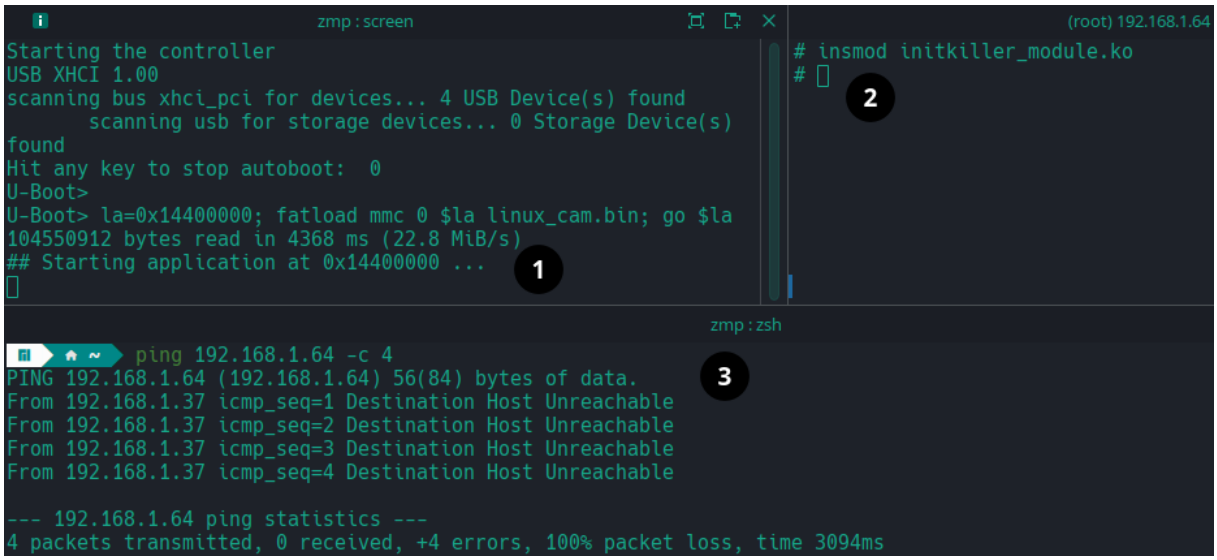


Figure 53: Functional tests: testing of the kill init process kernel module – USPFS

5.1.2 Unprivileged mode

User-space attacks present a reduced attack surface due to limited access to critical resources. Common approaches induce severe system contention, which—while not causing literal crashes—significantly degrades performance, potentially leading to catastrophic failure through critical system unresponsiveness.

Listing B.16 implements a comprehensive resource exhaustion application employing three parallel attack vectors: (1) memory exhaustion through infinite allocation while evading the OOM killer; (2) process exhaustion via a "fork bomb" generating processes that modify shared memory regions before hanging

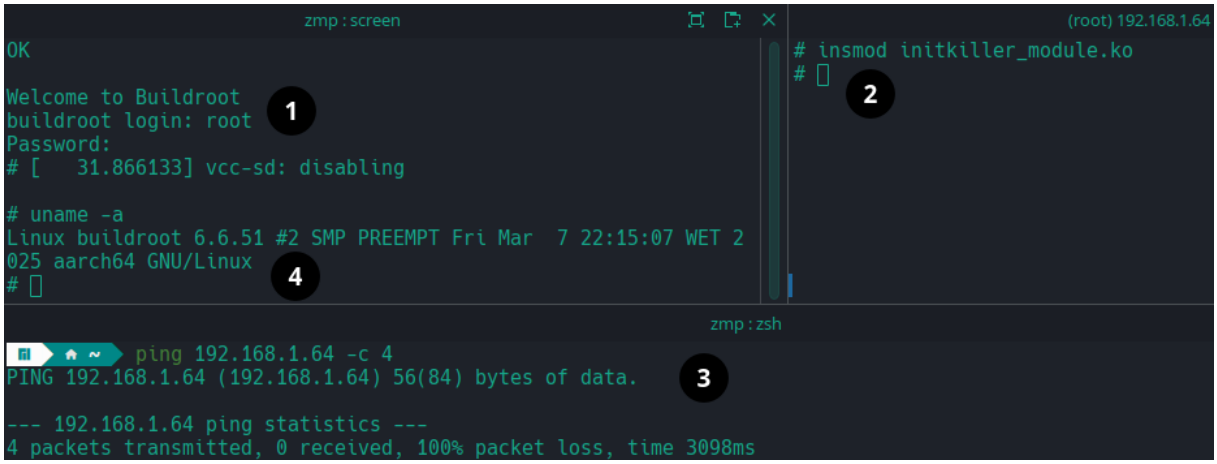


Figure 54: Functional tests: testing of the kill init process kernel module – SSPFS

(line 38); and (3) CPU/IO starvation achieved through highest scheduling priority assignment (line 45), continuous random page reads (line 63), and forced CPU rescheduling (line 56).

A monitoring thread (lines 97-131) tracks elapsed execution time, free memory (`sysinfo()`), process count (`sysinfo.procs`), CPU usage (`/proc/stat`), and load average (`sysinfo.loads`). The main function (lines 133-157) coordinates the attack by recording start time, removing resource limits, launching the monitoring thread, initiating three child processes for distinct attacks, and persisting via a non-optimizable infinite loop.

Fig. 55 demonstrates resource exhaustion testing in the USPFS system, showing: system boot via UART (top left, 1); remote application execution via SSH over Wi-Fi (top right, 2); system unresponsiveness evidenced by `ls` failure (3); and continued network presence despite functional failure (5).

Fig. 56 shows equivalent testing in the SSPFS system, featuring: shared UART boot console (Bao/PX4 VM, top left, 1); application execution in Companion VM (SSH over Wi-Fi, top right, 2); Companion VM unresponsiveness (`ls` failure, 3); maintained network presence (4); and unaffected PX4 VM operation (5) preserving UAV functionality.

These results confirm the Bao hypervisor's capacity to maintain critical flight functions during user-space resource exhaustion attacks, demonstrating effective consolidation with domain isolation.

5.2 Bao benchmarks

Previous research extensively benchmarked the Bao hypervisor against relevant competitors in MCS contexts [119]. This assessment adopts the same methodology while focusing exclusively on performance metrics.

The evaluation establishes baseline performance of the USPFS system (native execution) and quantifies performance degradation in the SSPFS system following Bao hypervisor introduction. Performance

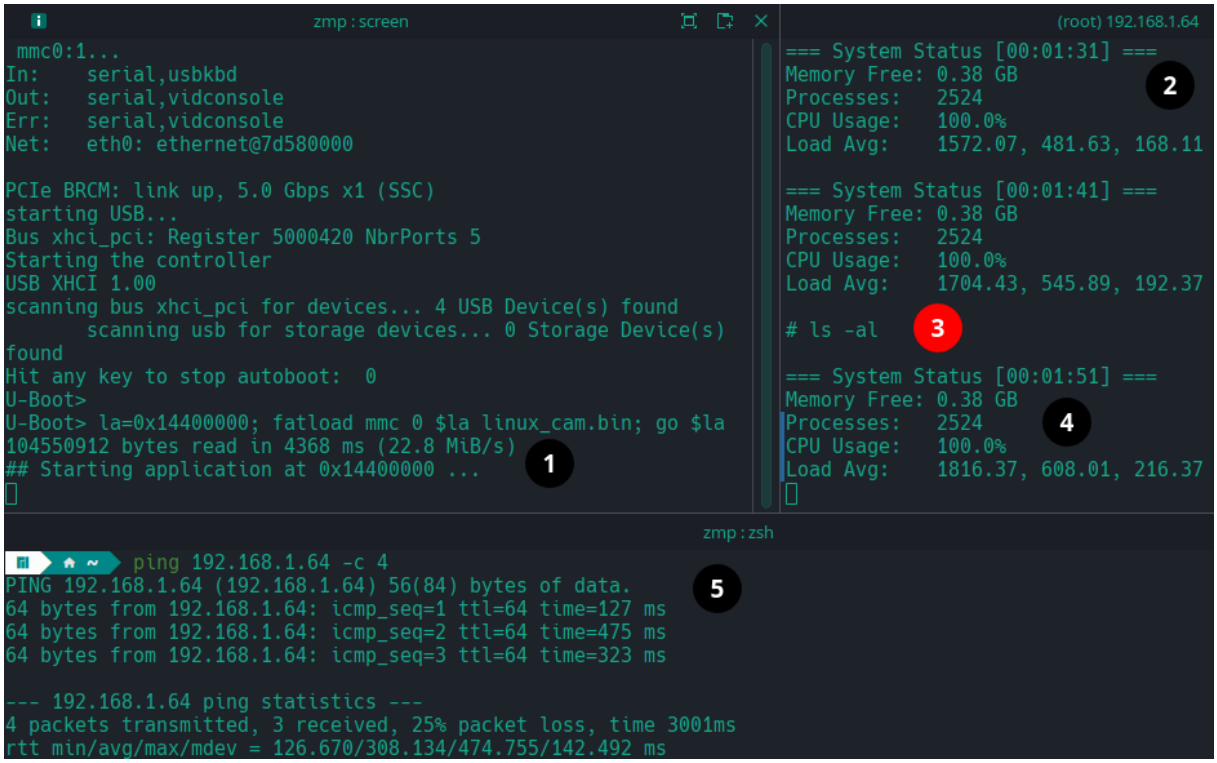


Figure 55: Functional tests: Resource exhaustion application testing – USPFS

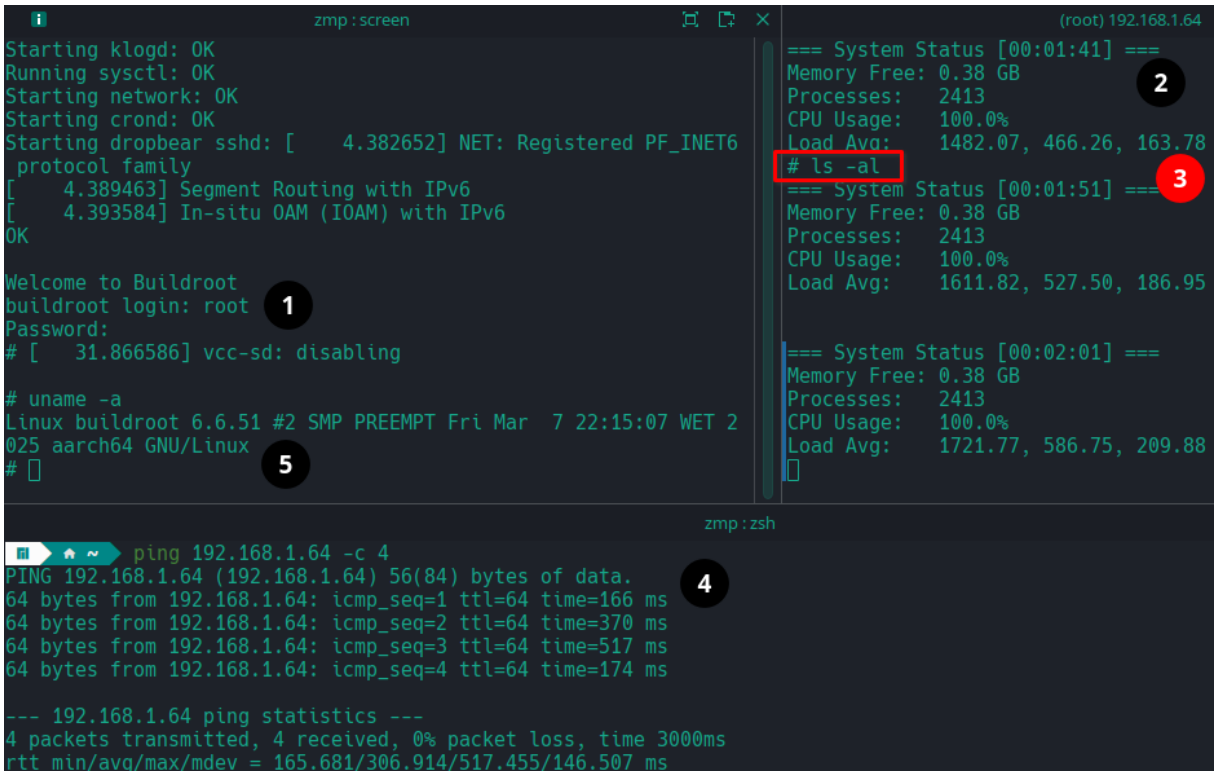


Figure 56: Functional tests: Resource exhaustion application testing – SSPFS

degradation is calculated as:

$$\text{Degradation} = \frac{\text{Execution time}_{\text{SSPFS}}}{\text{Execution time}_{\text{USPFS}}}$$

The testing framework utilizes the MiBench Embedded Benchmarks' AICS [120], featuring two variants: a small-scale variant with minimized input dataset replicating resource-constrained environments, and a large-scale variant with expanded dataset approximating real-world operational loads.

Measurement instrumentation employs `perf` [121] for timing and microarchitectural events in Linux VMs, while baremetal guests use the Arm Generic timer (10ns resolution) with a custom Performance Monitor Unit (PMU) driver. Monitored events include instruction count, TLB accesses/refills, cache accesses/refills, exceptions, and interrupts—all registered at their respective exception levels.

Interference assessment evaluates microarchitectural protection mechanisms through a custom baremetal guest with three Virtual CPUs (vCPUs) performing continuous write operations to a 1MiB buffer (matching LLC size). This workload represents significant though not worst-case interference.

Fig. 57 outlines the benchmarking workflow comprising three stages: guest construction, performance testing, and result logging with comparative analysis. Guest construction builds the Linux guest (MiBench suite + `perf`) and interference guest using SheddingLight repository instructions [122]. Performance testing evaluates the baseline USPFS and four SSPFS configurations: `mibench` (Linux guest + Bao), `mibench+col` (+ cache coloring), `mibench+interf` (+ interference guest), and `mibench+interf+col` (+ interference & coloring).

The experimental workflow executes sequentially: deployment of executable `BIN` (`linux.bin` or `bao.bin`) to SD card; establishment of persistent UART connection (green background) between GCS and UAVIC; power application to UAVIC platform; U-Boot initialization; execution of `BIN` with results logging; conditional execution of secondary binary `BIN2` via shell script with results redirection; capture file closure; and results visualization via Python plotting script.

Fig. 58 presents relative performance degradation for the MiBench AICS across three configurations: **bao** (MiBench on Linux OS + Bao hypervisor), **baremetal-noMail** (native execution without mailbox driver patch), and **bao-noMail** (Linux OS + Bao without mailbox driver patch). Average native execution times appear below each benchmark.

Performance degradation remains minimal, particularly for longer-running benchmarks (`large` suffix, e.g., `basicmath large`, `qsort large`). Key observations confirm the mailbox driver patch introduces negligible overhead, while the Bao hypervisor's contribution to performance degradation is statistically insignificant.

Fig. 59 presents relative performance degradation for the MiBench AICS under cache partitioning and interference across three configurations: **bao+col** (MiBench on VM1 with 2 cache colors), **bao+interf** (MiBench on VM1 with interference guest on VM2 using 3 CPUs), and **bao+interf+col** (MiBench on VM1 with 2 colors plus interference guest on VM2 using 3 CPUs and 2 colors). Cache colors were evenly distributed across virtual machines, with average native execution times provided below each benchmark.

Key observations reveal significant performance degradation under interference conditions, particularly affecting short-duration benchmarks (`small` suffix, e.g., `susanc small`, `susane small`). This degradation attenuates in longer-running benchmarks, while cache partitioning via page coloring consistently mitigates performance impacts for VM1.

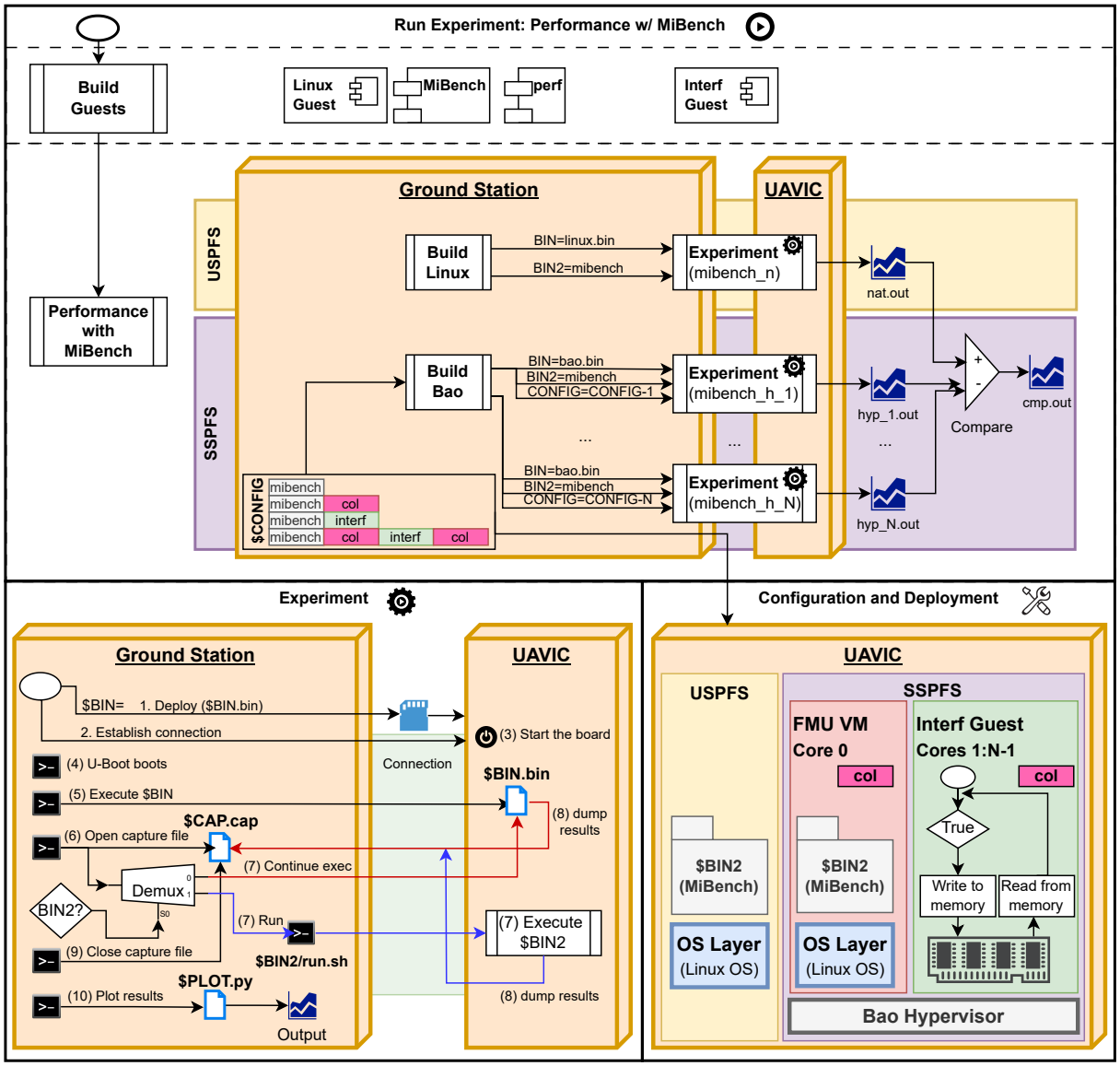


Figure 57: Bao performance benchmarking workflow

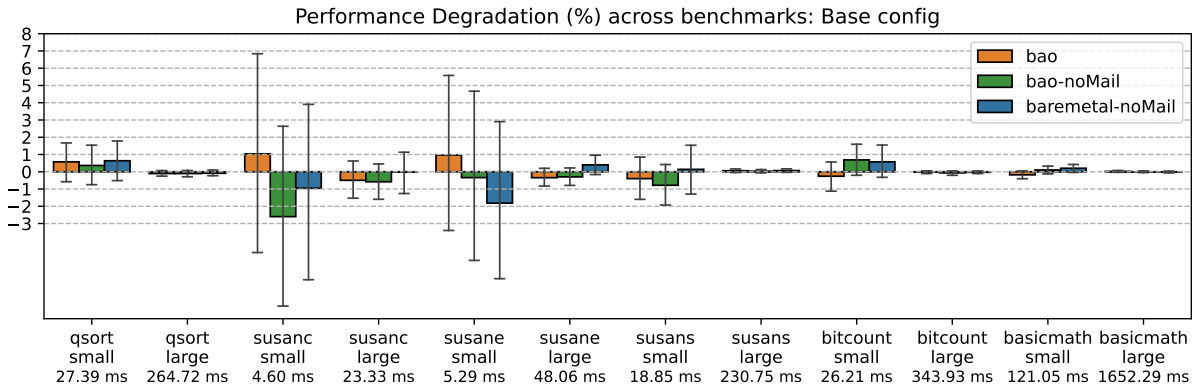


Figure 58: Relative performance degradation (%) for MiBench AICS

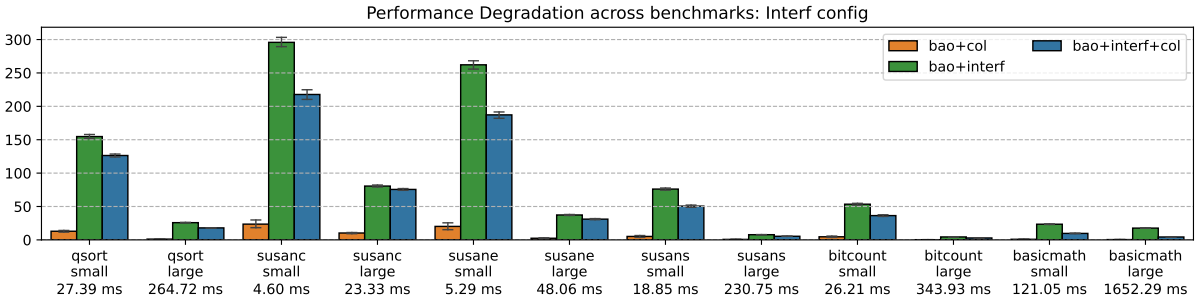


Figure 59: Relative performance degradation (%) for MiBench AICS under cache partitioning and interference

5.2.1 Guests benchmarking

This section presents guest benchmarking results for both single and dual-VM configurations.

5.2.1.1 FMU VM

The Flight Management Unit (FMU) VM executes the PX4 flight stack atop a Linux OS managed by the Bao hypervisor. The benchmarking methodology adapts previous approaches using `perf` for profiling, with key adaptations to accommodate PX4's continuous operation: 5 warm-up runs and 20 test runs of 30-second duration each. This time-limited execution strategy addresses the fundamental difference from finite-duration benchmarks like MiBench.

The PX4 benchmarking configuration (Listing B.17) includes random seed initialization, `perf` event tracking setup, and invocation of `px4_bench` with the `pilotpi_mc.config` profile. During execution, `perf` forks to run PX4 until the configured timeout while collecting performance data.

Execution time proved unsuitable for degradation measurement due to forced termination artifacts. Native execution averaged 30.044213 ± 0.000613 seconds, while Bao execution averaged 30.04608 ± 0.00199 seconds, yielding a maximum degradation of just 0.015

For critical embedded systems like the FMU, deadline compliance is paramount. PX4 supports two task models: traditional Tasks with separate stacks/priorities, and resource-efficient Work Queue Tasks that share stacks/priorities - the preferred approach for optimized resource utilization [123].

Task update rate analysis employed PX4's `work_queue` utility to monitor six critical flight operations:

1. `control_allocator` for actuator output generation;
2. `mc_rate_control` for multicopter rate management;
3. `pca9685_pwm_out` handling PWM-driven motor ESC via asynchronous I²C;
4. `flight_mode_manager` for flight mode setpoints;
5. `mc_pos_control` implementing position/velocity control; and
6. `sensors` for data acquisition and publishing [124–126].

The experimental methodology comprised 5 warm-up runs followed by 20 test runs. Each test run collected `work_queue` metrics at 10-second intervals over a 3-minute duration, totaling 18 queries per run. This process was executed on both native and Bao hypervisor configurations to enable comparative analysis.

Fig. 60 shows the results of the experiments. In the x-axis are indicated the work queue tasks and the mean update interval for the native execution in microseconds (baseline). The bar plots indicate the mean scheduling overhead (%) and standard deviation for the Bao case. The scheduling overhead is defined as the relative average overhead between the Bao and native executions, given by Equation 5.1 and Equation 5.2:

$$\mu_{ov} = \frac{\mu_{bao} - \mu_{bm}}{\mu_{bm}} \cdot 100 \tag{5.1}$$

$$\sigma_{ov} = \sqrt{\left(\frac{\sigma_{bao}}{\mu_{bao}}\right)^2 + \left(\frac{\sigma_{bm}}{\mu_{bm}}\right)^2} \cdot 100 \tag{5.2}$$

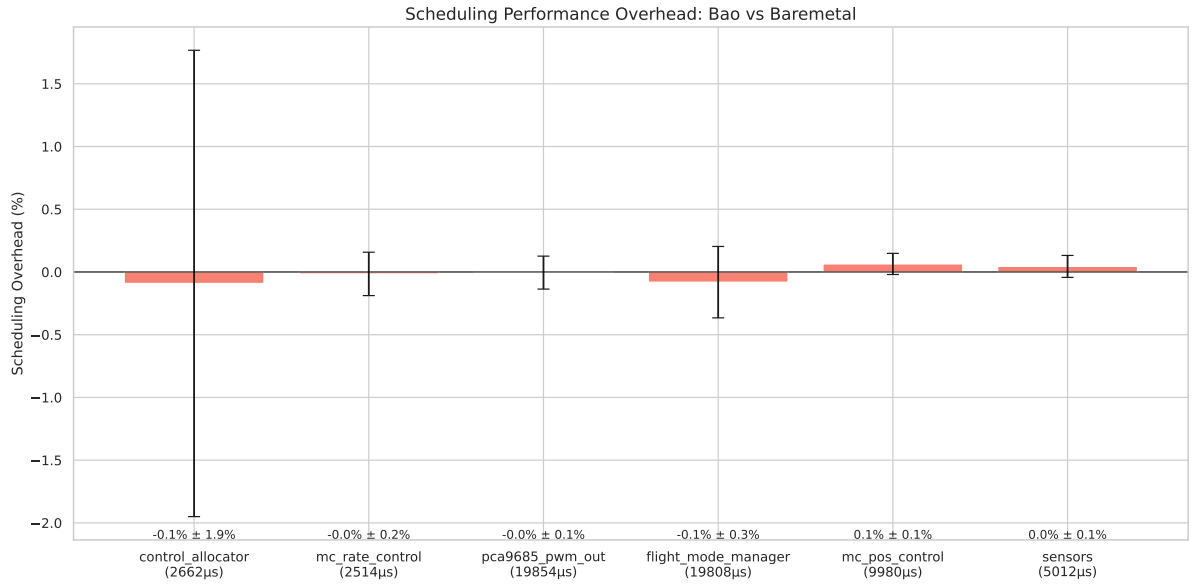


Figure 60: PX4 tasks scheduling overhead

The scheduling overhead is very low across work queue tasks. Most notably, the `pca9685_pwm_out` which is ultimately responsible for keeping the UAV on the air, has a negligible scheduling overhead. On the other hand, the `control_allocator`, responsible for managing the control loop, has a scheduling overhead of at most 2%. These results demonstrate the consolidation of the software stacks into a single platform, enabled by the Bao hypervisor, imposes minimum scheduling overhead in the critical system (FMU).

5.2.1.2 Companion VM

The Companion VM serves as the core component for video surveillance operations, employing a `gstreamer` pipeline that manages frame acquisition, decoding/encoding processes, packetization routines, and UDP-based data transmission. User experience constitutes a critical factor in evaluating video surveillance service quality, with particular emphasis placed on two key performance metrics: the receiver's FPS and frame latency – defined as the temporal delay between frame transmission and reception. These metrics collectively determine whether the system delivers a perceptually *real-time* viewing experience. However, system performance may be significantly degraded by various external factors, particularly fluctuations in GCS CPU utilization and network latency variations.

To objectively quantify the performance implications of platform consolidation, this study employs frame rate analysis conducted at the UAV level. This methodological approach isolates the effects of system integration from external environmental variables. This analysis comprised 5 warm-up runs and 20 test runs of the `gstreamer` pipeline for two minutes each.

Fig. 61 presents the relative FPS performance degradation between Bao and native execution across runs. The performance degradation is calculated using Equation 5.1 and Equation 5.2. Most runs show degradation close to 0%, with values ranging between -0.2% to +0.4%, and top degradation is inferior to 2%. The confidence intervals are narrow, indicating precise measurements, and they include 0%, meaning Bao introduces no statistically significant overhead to the UAV's camera frame rate.

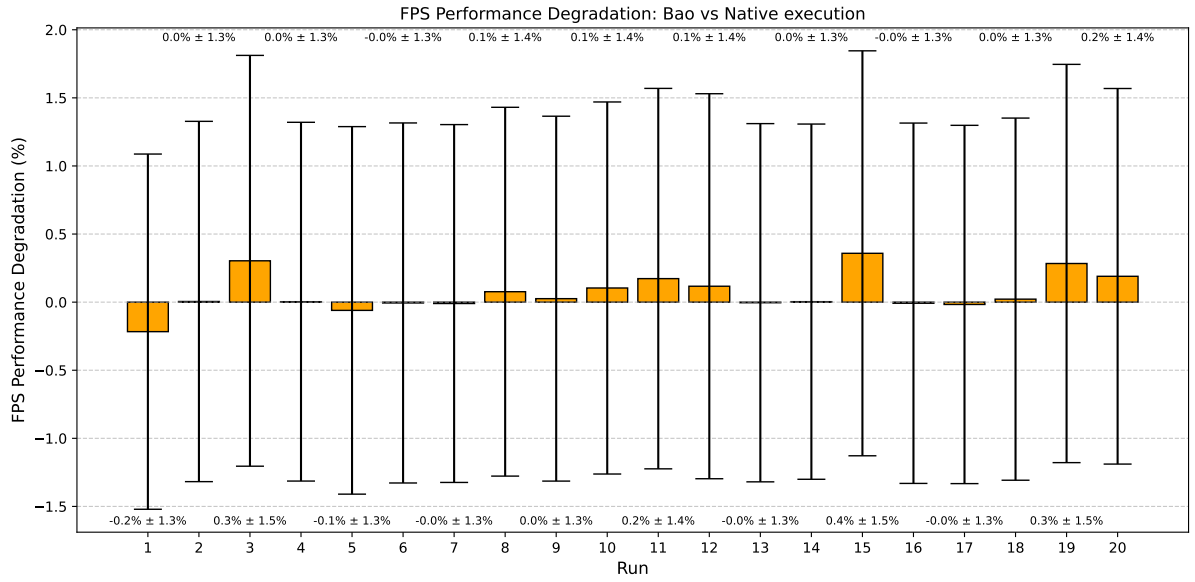


Figure 61: Relative FPS Performance Degradation: Bao vs native execution

5.2.2 USPFS vs SSPFS

Following individual guest benchmarking, this analysis compares consolidated system performance between the unsupervised (USPFS) and supervised (SSPFS) architectures. The evaluation employed: (1)

PX4 scheduling overhead tests for the flight management unit; camera's frame rate (FPS) tests for video surveillance

Both tests executed concurrently on each system, with data logging configured appropriately for each architecture's process/VM execution model.

PX4 Scheduling Performance Fig. 62 contrasts scheduling overhead between architectures. Key observations:

- SSPFS shows marginal degradation (1%) in `flight_mode_manager` and `pca9685_pwm_out` tasks
- SSPFS outperforms USPFS in other tasks, notably `control_allocator` (6-15% improvement)

This performance divergence likely stems from Bao's isolation guarantees: while USPFS processes contend for shared resources, SSPFS VMs operate within dedicated resource partitions.

Cache coloring provided negligible benefits due to DMA constraints requiring specific physical memory mapping (Listing C.7), which conflicts with coloring's virtual memory optimizations.

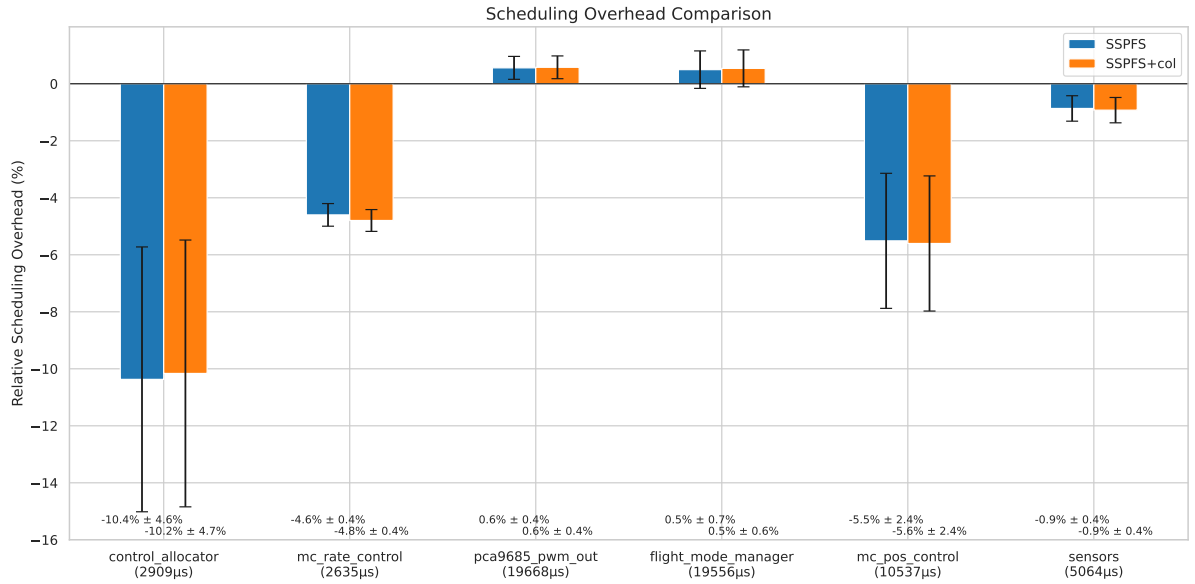


Figure 62: PX4 tasks scheduling overhead: USPFS vs SSPFS

Video Performance Fig. 63 presents relative FPS degradation. Results indicate:

- No statistically significant overhead in most SSPFS runs
- Outlier improvements exceeding 30% in runs 10/20
- Cache coloring again negligible due to DMA constraints

These findings align with previous Companion VM benchmarks, confirming minimal video impact.

Conclusion Comparative analysis demonstrates that beyond isolation benefits, the supervised SSPFS architecture can yield performance advantages in consolidated mixed-criticality systems. Static resource

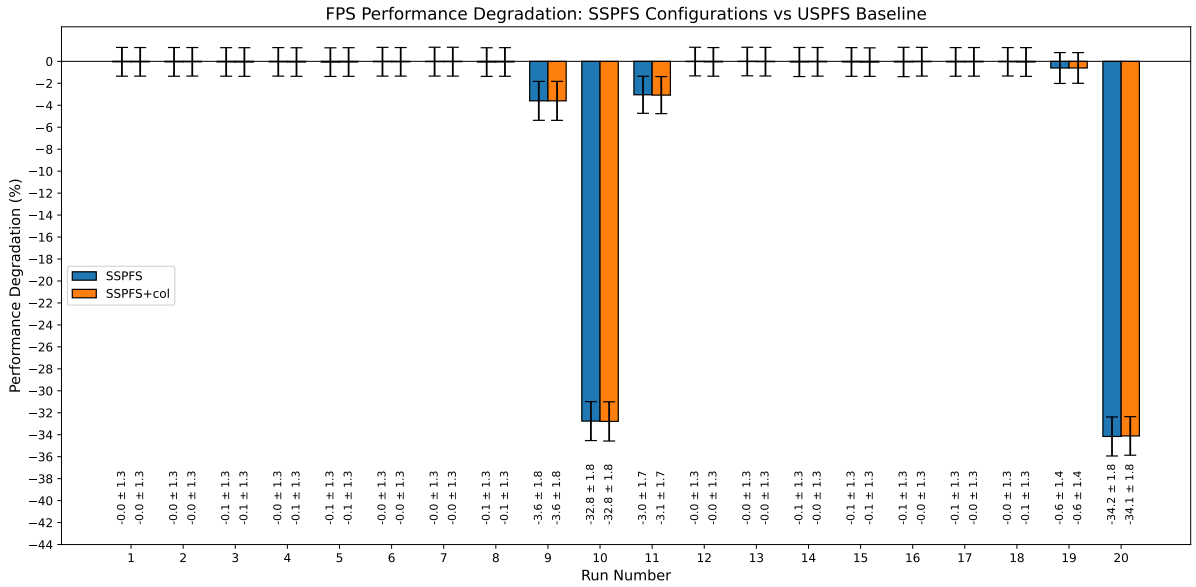


Figure 63: Relative FPS Performance Degradation: USPFS vs SSPFS

partitioning eliminates contention-related overheads observed in USPFS, establishing SSPFS as the optimal solution for flight controller and video surveillance integration.

5.3 UAV benchmarks

To evaluate both flight stacks under realistic conditions, an automated mission was configured in QGroundControl (Fig. 64). This standardized flight profile ensured repeatable testing across the USPFS and SSPFS systems while benchmarking UAV performance in operational scenarios.

The mission design incorporates two critical geospatial elements: an external polygon defining the geofence perimeter, which triggers a return-to-home failsafe if violated; and an internal flight path beginning at takeoff point T (ascending to 2m altitude). The route proceeds sequentially to waypoint 3, reverses direction to 4, and concludes at landing point L, maintaining constant altitude throughout.

Experimental execution comprised two batches of 33 flights per system, conducted under consistent weather conditions. The USPFS system flights preceded SSPFS testing due to practical constraints: system switching required significant battery and time resources, preventing randomized sequencing. All flight logs were preserved for subsequent analysis of two critical metrics: (1) position tracking accuracy, and (2) system resource utilization patterns. The sample size determination prioritized statistical power to detect meaningful differences in these operational parameters. To account for temporal variations between flights, the time domain was normalized to a mission progress scale $[0, 100]$ using piecewise cubic interpolation:

$$t_{\text{norm}} = \frac{t - t_{\min}}{t_{\max} - t_{\min}} \times 100 \tag{5.3}$$

Figure 64: Automated mission configuration in QGroundControl

where t is the original timestamp, and $t_{\min}$ and $t_{\max}$ represent the start and end times of each flight respectively.

At each normalized time point τ, the 95% confidence intervals for group means were calculated using the Student's t-distribution:

$$\text{CI}(\tau) = \bar{x}(\tau) \pm t_{\alpha/2,df} \cdot \frac{s(\tau)}{\sqrt{n(\tau)}} \tag{5.4}$$

where:

- $\bar{x}(\tau)$ is the sample mean at progress τ
- $s(\tau)$ is the sample standard deviation
- $n(\tau)$ is the number of valid observations
- $df = n(\tau) - 1$ degrees of freedom
- $t_{\alpha/2,df}$ is the critical t-value ($\alpha = 0.05$)

Statistical significance between systems was evaluated using Welch's unequal variance t-test at each mission progress point:

$$t(\tau) = \frac{\bar{x}_1(\tau) - \bar{x}_2(\tau)}{\sqrt{\frac{s_1^2(\tau)}{n_1(\tau)} + \frac{s_2^2(\tau)}{n_2(\tau)}}} \tag{5.5}$$

with degrees of freedom approximated by the Welch-Satterthwaite equation:

$$df(\tau) \approx \frac{\left(\frac{s_1^2(\tau)}{n_1(\tau)} + \frac{s_2^2(\tau)}{n_2(\tau)}\right)^2}{\frac{s_1^4(\tau)}{n_1^2(\tau)(n_1(\tau)-1)} + \frac{s_2^4(\tau)}{n_2^2(\tau)(n_2(\tau)-1)}} \tag{5.6}$$

The null hypothesis (H_0) states no difference between systems:

$$H_0 : \mu_{\text{USPFS}}(\tau) = \mu_{\text{SSPFS}}(\tau) \tag{5.7}$$

H_0 was rejected at $\alpha = 0.05$ significance level when $p(\tau) < 0.05$.

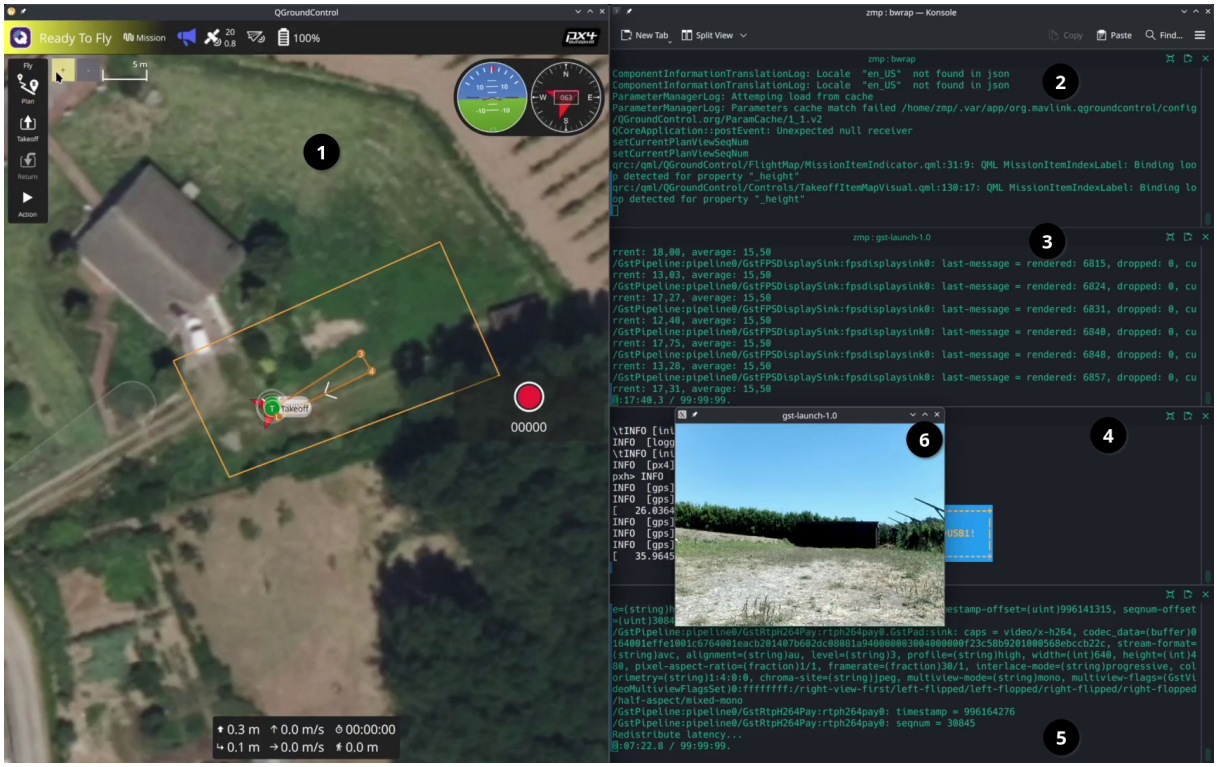
Relevant data were extracted from PX4 flight logs stored as ULog files (`.ulg`), containing uORB topic messages subscribed to or published by various modules. Specifically, the `vehicle_local` and `vehicle_local_setpoint` topics were analyzed for position tracking assessment, while the `cpuload` topic was examined for system resource utilization evaluation.

Mission execution during critical phases is illustrated for the SSPFS system in Fig. 65, with the detailed mission plan presented in Fig. 66. Fig. 65a depicts the take-off phase alongside the employed software components, including QGroundControl for mission flight management and logging on the host system, the `gstreamer` receiver pipeline with its associated Graphical User Interface (GUI) executing on the host, the U-Boot loader, and the FMU VM console responsible for PX4 execution. The `Companion` VM, which runs the `gstreamer` sender pipeline, was accessed via Secure Shell (SSH) over Wi-Fi. Flight path progression through intermediate stages and corresponding video streaming during flight operations are shown in Fig. 65b and Fig. 65c. The landing phase, featuring the UAV operator, is visible in Fig. 65c, while Fig. 65d demonstrates successful mission completion. These results confirm that the SSPFS system functions as designed: the autopilot operates within the FMU VM, exchanging data with the UAV through the telemetry radio link, while video streaming executes on the `Companion` VM, capturing live feed from the USB camera and transmitting it over Wi-Fi.

To interpret the test results, analysis of the actual mission flight path – illustrated as a red line in Fig. 66 - is essential. Mission paths exhibit slight variations due to differing sensor estimates and weather conditions. As anticipated, the autopilot dynamically adjusts trajectories according to vehicle flight dynamics and parameters rather than strictly following predefined paths. This adaptation is particularly evident during the sharp turn between points `3` and `4`. Consequently, although PX4 anticipates linear progression between waypoints, it modulates velocity during approach and departure phases based on jerk-limited tuning parameters, while adapting the trajectory to follow smooth curves defined by the acceptance radius parameter (`NAV_ACC_RAD`) [127].

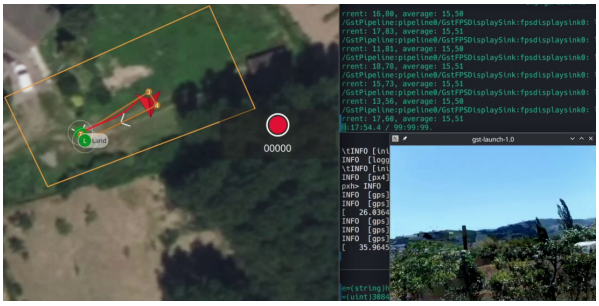
Position tracking performance comparison between the USPFS and SSPFS systems across X, Y, and Z axes is presented in Fig. 67. Dotted lines indicate setpoint trajectories, while blue and orange traces represent USPFS and SSPFS systems respectively, with shaded bands denoting 95% confidence intervals.

The controller demonstrates accurate position tracking, with observed trajectories closely following setpoints except during take-off and landing phases due to flight dynamics. Notably, setpoint variations



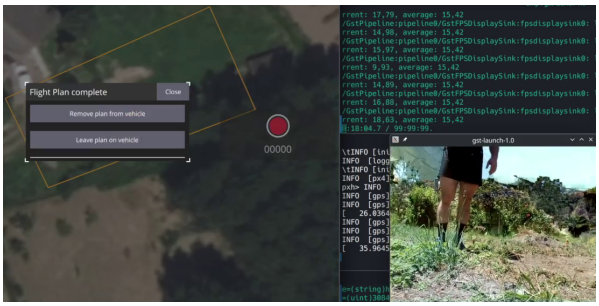
(a) Take-off

(b) Point 3



(c) Point 4

(d) Landing



(e) Complete

Figure 65: Mission execution – SSPFS case

occur between systems despite identical missions, attributable to differing flight dynamics influenced by environmental factors—particularly air pressure variations affecting Z-axis estimates—and sensor estimation discrepancies, such as inherent GPS noise and temporal drift.

Significant band overlap indicates no statistical difference in X and Z-axis tracking, though a consistent

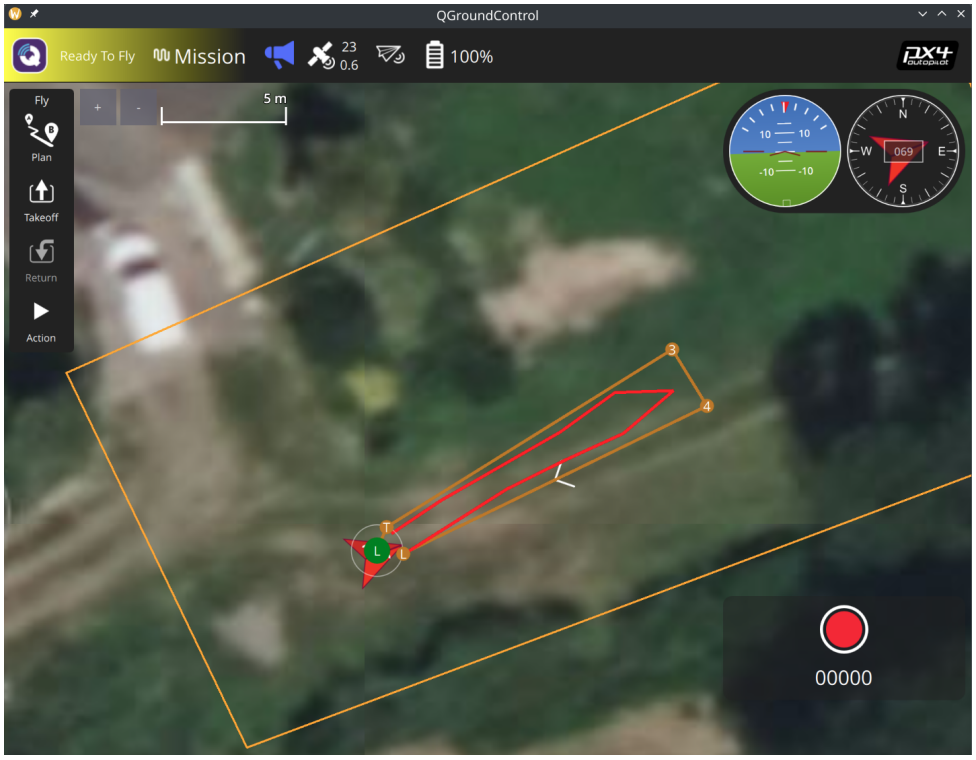


Figure 66: Actual flight path of a mission in QGroundControl

Y-axis offset of approximately 2 meters is observed. Further data collection is required to determine the causal factors for this deviation. Crucially, supervisory mechanisms in the autopilot demonstrate no adverse impact on overall position tracking performance.

System resource utilization comparison, derived from PX4 uORB messages rather than direct system measurements, appears in Fig. 68. The supervised system introduces measurable overhead: 6% increase in CPU load and 9-fold RAM utilization growth. CPU results align with prior Bao benchmarks (Section 5.2), while RAM expansion may stem from firmware mailbox supervision requiring Bao's mailbox manager to intercept transactions. Validation would require alternative device-sharing mechanisms between VMs, currently unavailable. However, this RAM increase remains negligible given the FMU VM's 144 MB allocation.

Functional tests were replicated in real-flight scenarios. System behavior during `Companion` VM compromise is documented in Fig. 69 through Fig. 71, with upper subfigures showing UAV perspectives and lower subfigures displaying operator views.

Fig. 69 captures post-takeoff operations and associated software components: QGroundControl mission planning and host logging; host-executed `gstreamer` receiver pipeline and GUI; telemetry-transmitted system diagnostics via PX4 console; U-Boot loader and FMU VM executing PX4; SSH-accessed `Companion` VM running `gstreamer` sender pipeline; and remote SSH shell for VM compromise. Video streaming operates nominally during initial takeoff.

Fig. 70 captures the state following execution of the malicious kernel module within the `Companion` VM. Video streaming immediately freezes, indicating compromised functionality in this virtual machine.

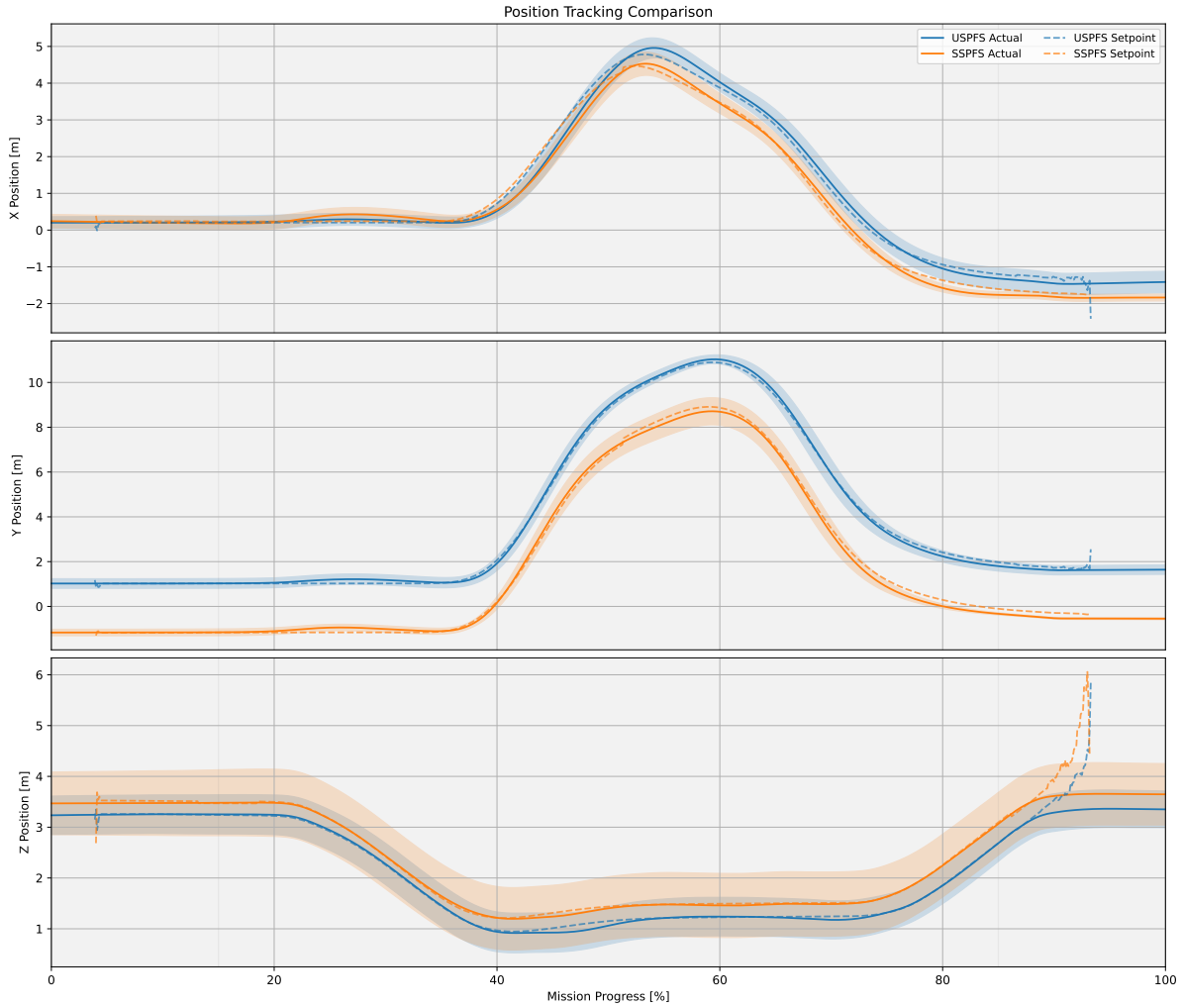


Figure 67: Position tracking comparison between USPFS and SSPFS systems

Despite this failure, mission completion is confirmed by both the QGroundControl dialog box (Fig. 71a) and operator perspective (Fig. 71b), where safe UAV landing is observable. These results demonstrate Bao's hypervisor capability to effectively isolate the critical flight stack (FMU) from non-critical components (`Companion`). Consequently, crashes in the `Companion` VM remain contained without propagating to the autopilot, thereby preventing potential catastrophic UAV failure.

The identical functional test procedure was subsequently applied to the USPFS system. System behavior during compromise is documented in Fig. 72 through Fig. 74, with upper subfigures presenting UAV perspectives and lower subfigures showing operator viewpoints.

Fig. 72 illustrates post-takeoff operations and associated software components: QGroundControl mission planning and logging executing on the host system; host-based `gstreamer` receiver pipeline with its GUI; remote SSH shells managing the `gstreamer` sender pipeline, PX4 execution monitoring, and system compromise initiation. Video streaming functions nominally during initial takeoff.

Fig. 73 captures system state following malicious kernel module execution within the USPFS environment. Video streaming cessation indicates system failure, accompanied by uncontrolled UAV maneuvers

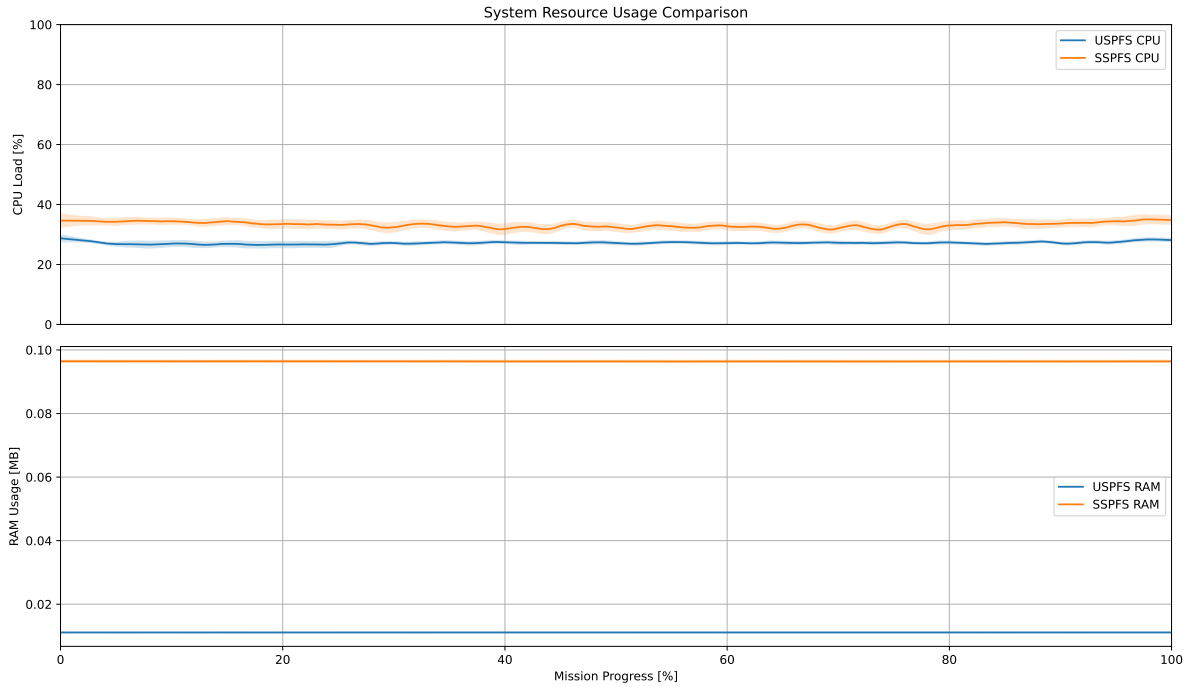


Figure 68: System resource usage comparison between USPFS and SSPFS systems

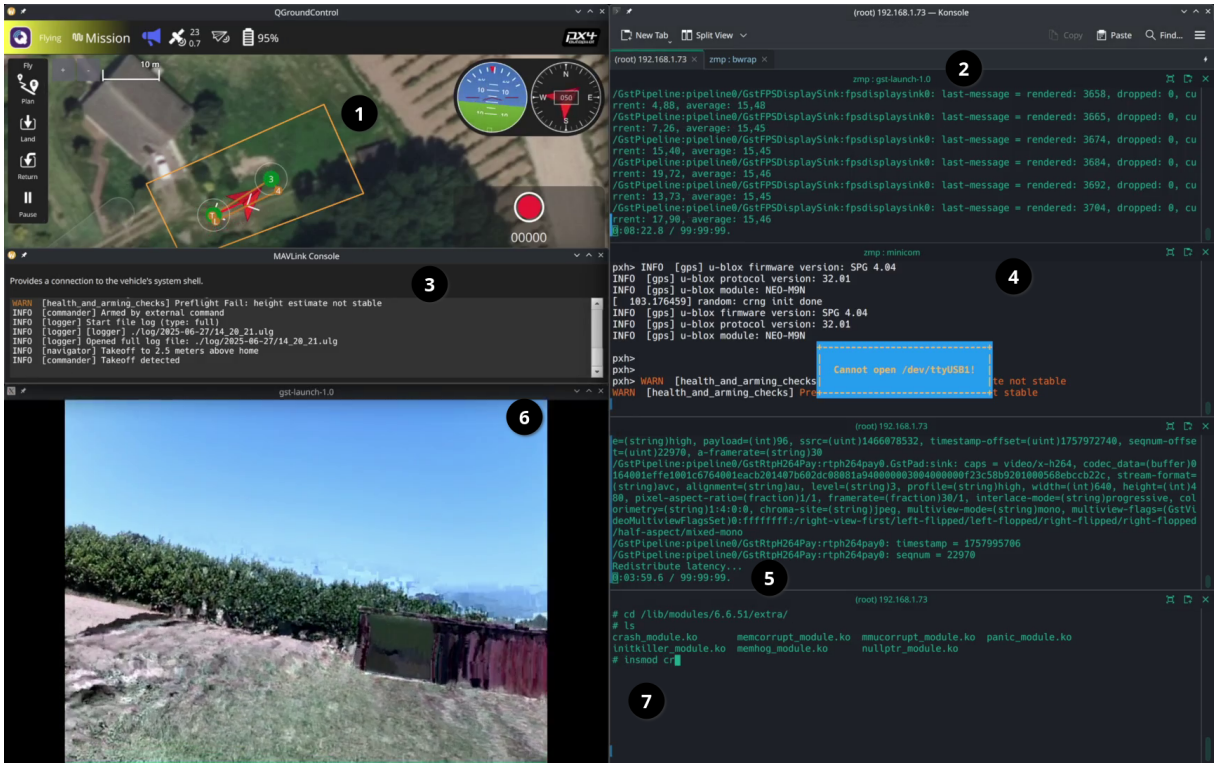
including flipping motions.

Fig. 74 documents the UAV ground impact event coinciding with `QGroundControl` connection loss to the USPFS platform. This outcome demonstrates that in unsupervised architectures, non-critical component failures propagate system-wide, resulting in catastrophic UAV loss.

5.4 Summary

This chapter presented testing and evaluation of the proposed solution. Functional testing analyzed unsupervised (USPFS) and supervised (SSPFS) system behavior under compromise scenarios. Multiple attack approaches were examined across privileged and unprivileged execution modes. Privileged compromise enables system hijacking through malicious kernel modules inducing system halting (`panic`), memory corruption, resource exhaustion, or termination of the `init` process. Unprivileged attacks are constrained to user-space exploits, substantially reducing the attack surface, typically manifesting as resource contention that significantly degrades system performance. USPFS compromise causes simultaneous failure of critical FMU and non-critical video surveillance stacks, resulting in total system failure. Conversely, SSPFS containment prevents non-critical stack failures from propagating to the critical domain, maintaining UAV operation despite video streaming interruption. These results demonstrate Bao hypervisor's capability to consolidate mixed-criticality software stacks on a single platform while ensuring domain isolation.

Benchmarking compared USPFS baseline performance against SSPFS degradation introduced by the Bao hypervisor using the MiBench Automotive and Industrial Control Suite (AICS) suite. Analysis included



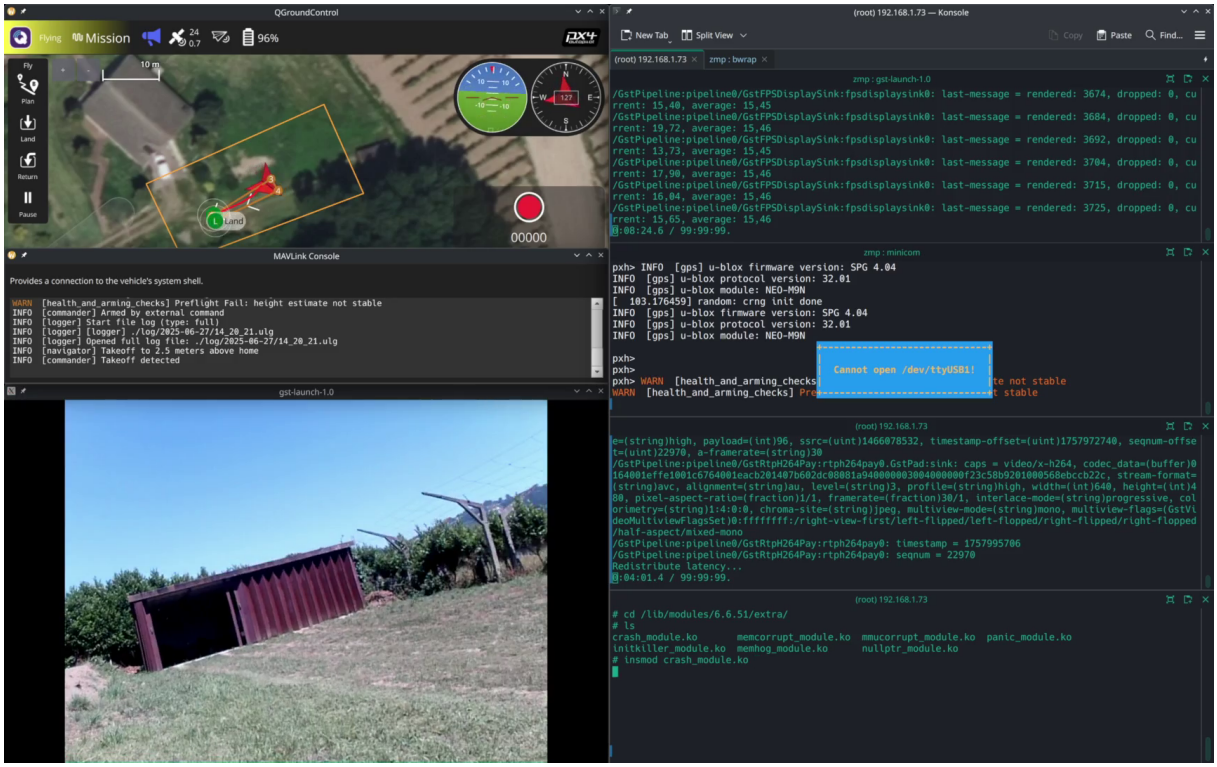
(a) UAV's view

(b) Operator's view

Figure 69: Mission execution: Functional test (SSPFS) – Take-off

guest interference effects and Raspberry Pi firmware mailbox driver modifications. Results indicate negligible performance degradation attributable to either Bao or mailbox patches. However, significant performance degradation occurs under interference conditions, particularly for shorter benchmarks, though this effect diminishes over execution duration. Cache partitioning via page coloring consistently reduces VM1 performance degradation, demonstrating utility for interference mitigation.

Guest benchmarking in single- and dual-VM configurations evaluated PX4 task scheduling overhead and camera frame rates. Bao introduces minimal scheduling overhead (≤2% worst-case) in the critical system and statistically insignificant frame rate impact. Comparative analysis reveals that beyond isolation benefits, the supervised SSPFS solution offers potential performance advantages through static resource



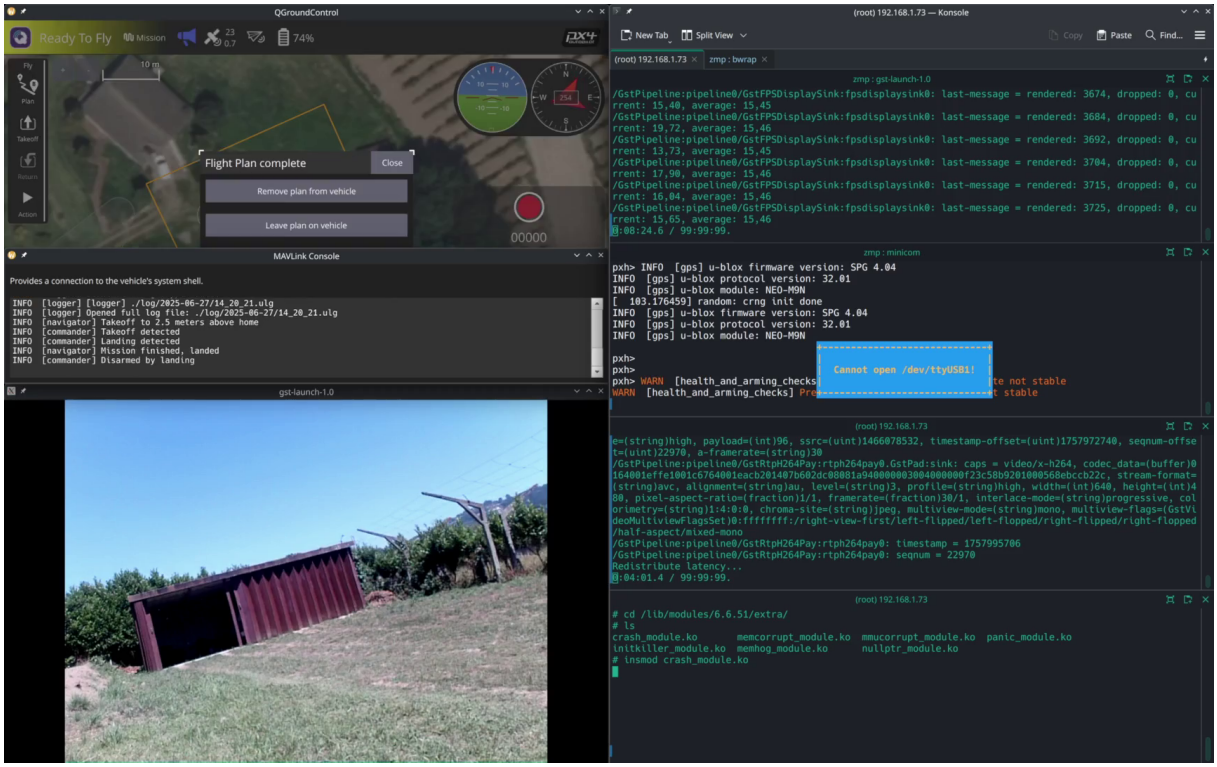
(a) UAV's view

(b) Operator's view

Figure 70: Mission execution: Functional test (SSPFS) – executing a malicious kernel module

partitioning.

Real-flight evaluation employed automated missions to compare position tracking and resource utilization. Supervision introduction shows no position tracking degradation, with modest CPU overhead (6% average). Significant RAM expansion (99 KB) is attributed to firmware mailbox supervision requiring Bao's mailbox manager transaction handling. Verification would require alternative VM device-sharing mechanisms, currently unavailable. This RAM increase remains operationally negligible given the FMU VM's 144 MB allocation. Repeated functional tests confirmed critical stack resilience during non-critical failures. Consequently, SSPFS emerges as the optimal solution for consolidating mixed-criticality FMU and video

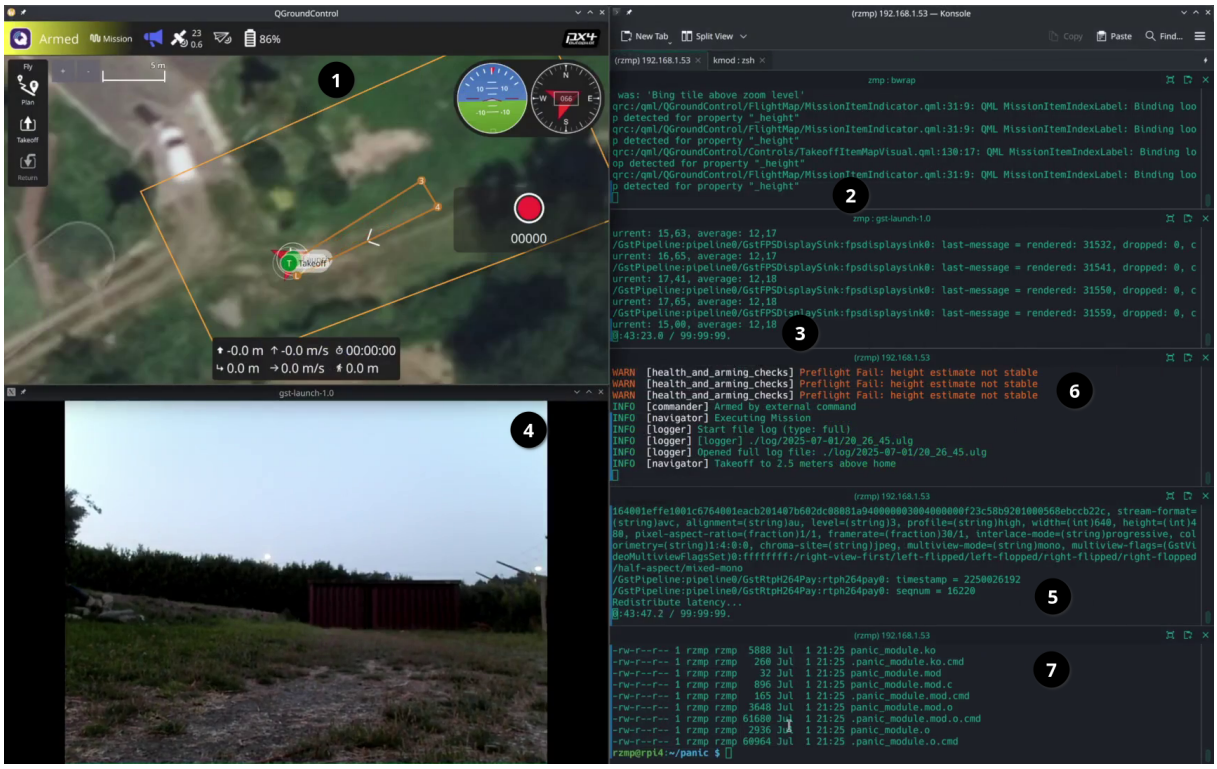


(a) UAV's view

(b) Operator's view

Figure 71: Mission execution: Functional test (SSPFS) – mission complete

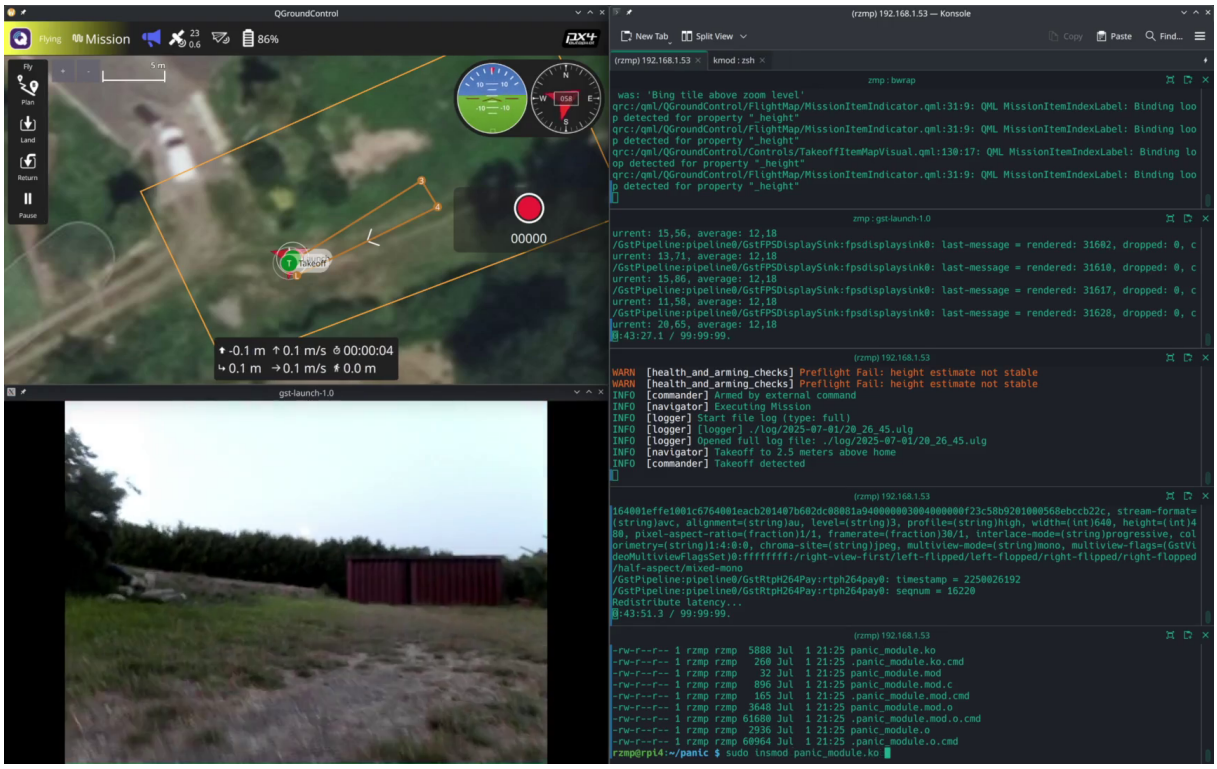
surveillance stacks, providing robust isolation guarantees with minimal performance overhead.



(a) UAV's view

(b) Operator's view

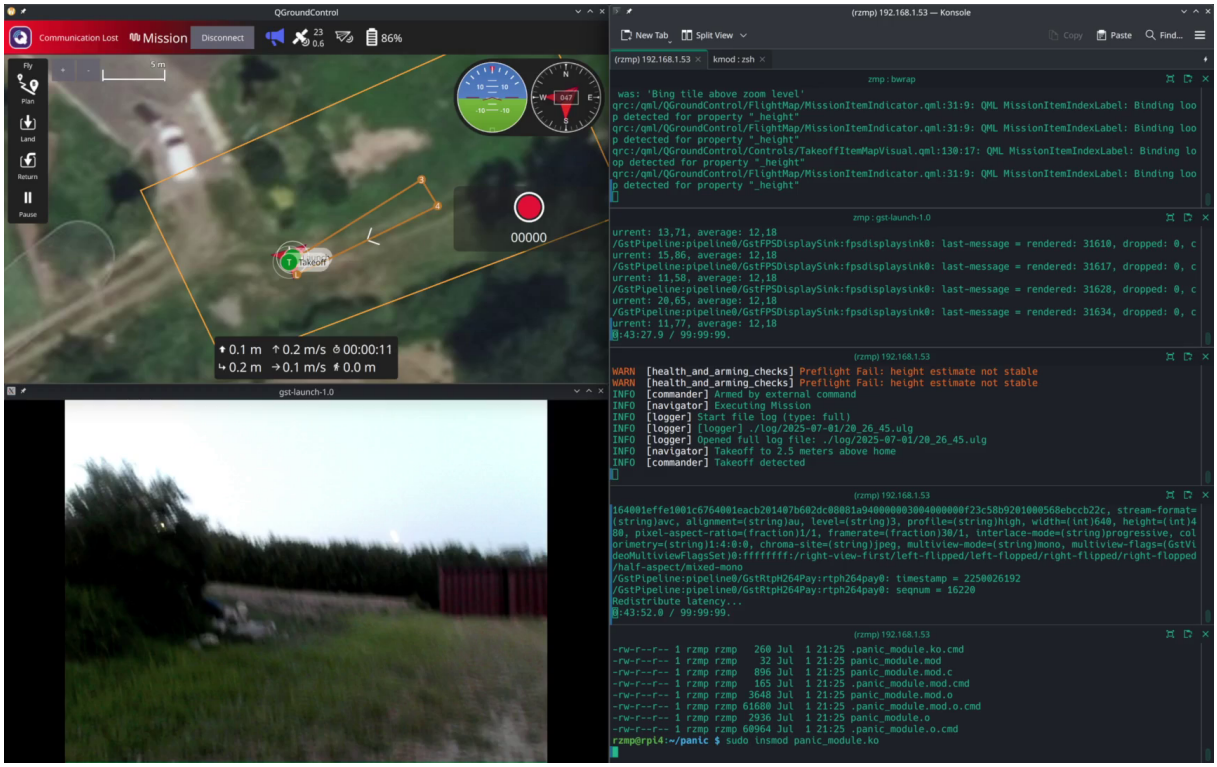
Figure 72: Mission execution: Functional test (USPFS) – take-off



(a) UAV's view

(b) Operator's view

Figure 73: Mission execution: Functional test (USPFS) – executing a malicious kernel module



(a) UAV's view

(b) Operator's view

Figure 74: Mission execution: Functional test (USPFS) – UAV's crash

6

Conclusions and Future Work

"The only way of discovering the limits of the possible is to venture a little way past them into the impossible."
– Arthur C. Clarke, science fiction writer and inventor

In this chapter the main conclusions about the present work are outlined, as well as the prospects for future work.

6.1 Conclusions

The work presented focused on the development of a trustworthy open-source software stack for UAV applications, with a paramount emphasis on security and safety. This effort aimed to bridge the gap between existing open-source and commercial solutions by contributing to the widespread adoption of secure and safe features in UAV technology. A key finding was that conventional multi-platform flight stacks, while addressing mixed-criticality at the hardware level, undesirably increase the UAV's weight and footprint. Furthermore, such systems typically lack isolation guarantees, meaning that a compromise in a non-critical component, like video surveillance, could propagate to the critical flight controller, potentially leading to catastrophic failure.

To overcome these limitations, the proposed Supervised Single-Platform Flight Stack (SSPFS) integrates the flight controller and companion computer functionalities onto a single hardware platform, leveraging the Bao hypervisor. The unsupervised counterpart solution – Unsupervised Single-Platform Flight Stack (USPFS) – was devised and tested to assess the impact of the supervision layer. The UAVIC platform selected consisted of a Raspberry Pi 4 + PilotPi shield, supporting the open source PX4 autopilot flight stack. The hardware mapping revealed the need to support a supervised mailbox access to handle firmware's mailbox transactions in the Raspberry Pi as a means to share some critical devices in the UAVIC platform, such as the PCIe bus.

The evaluation of the SSPFS demonstrated several critical advantages:

- **Enhanced Isolation**: Functional tests clearly showed that the Bao hypervisor provides strong isolation guarantees between mixed-criticality systems. When a malicious kernel module or a resource exhaustion application was introduced into the non-critical Companion VM, it caused that VM to crash, but the critical PX4 Flight Management Unit (FMU) VM remained operational and the UAV stayed airborne. In contrast, the unsupervised system suffered a total collapse under similar attacks.

- **Extensive benchmarking**: The supervised solution was benchmarked using the industrial standard MiBench suite, showing a very low performance degradation compared to the unsupervised one. The performance degradation can be significantly higher under interference, but cache partitioning via page coloring consistently reduced the performance degradation for the FMU VM, mitigating this effect. **Effective Mailbox Supervision**: A customized mechanism was devised and validated to manage shared access to the firmware mailbox between different VMs under Bao's supervision. Although not ideal and specific to the UAVIC platform selected – the Raspberry Pi 4 – this mechanism ensured seamless communication without compromising isolation.

- **Minimal Performance Overhead**: Benchmarking revealed negligible performance degradation due to the introduction of the Bao hypervisor and the custom mailbox driver patch required for Raspberry Pi firmware interfacing. For PX4 tasks, the scheduling overhead was found to be very low, with a worst-case increase of only 2%. The camera's frame rate in the Companion VM also showed no statistically significant overhead, with most runs exhibiting degradation close to 0% and a maximum of less than 2%. In some instances, the SSPFS even outperformed the unsupervised system due to the static partitioning of resources.

- **Real-Flight Validation**: Automated missions in a real-flight scenario confirmed that the SSPFS system maintains accurate position tracking, unaffected by the supervision layer. The CPU load increase was small (an average of 6%), which is in line with the benchmarkings performed on the Bao hypervisor. While an increase in RAM usage was observed (99 KB), it was considered negligible given the available memory for the FMU VM (144 MB).

In summary, the SSPFS system, leveraging the Bao hypervisor, has been demonstrated as a superior solution for consolidating mixed-criticality software stacks in UAVs, offering robust isolation guarantees with low performance overhead.

6.2 Future Work

Based on the insights gained from this work, several avenues for future research and development can be pursued:

- **Support for different UAVIC platforms:** the consolidation of the mixed-criticality stacks relied on a Raspberry Pi with the Linux GPOS. Ideally, the UAVIC platform should support the NuttX RTOS, required by the PX4 autopilot, and have a smaller footprint.
- **Advanced Shared Resource Management:** While cache coloring was explored, its effectiveness was limited in certain scenarios due to hardware constraints like DMA transactions. Future work should investigate and implement more sophisticated state-of-the-art partitioning mechanisms, such as memory throttling, to further mitigate interference from shared hardware resources like LLCs and interconnects, which can breach temporal isolation.
- **Optimized Interrupt Virtualization:** The current Bao hypervisor implementation supports Arm GICv2 and GICv3, which necessitates the hypervisor to re-inject interrupts into guest VMs. Future work could focus on updating Bao to fully leverage the GICv4 specification, which bypasses the hypervisor for guest interrupt delivery, potentially leading to further reductions in interrupt latency and overall complexity of interrupt management.
- **Detailed RAM Usage Analysis and Optimization:** The observed increase in RAM usage attributed to the firmware's mailbox supervision warrants deeper investigation. Exploring alternative mechanisms for securely sharing devices across VMs could provide further insights and potentially lead to more memory-efficient solutions.
- **Extended Functional and Performance Testing:** While initial functional tests demonstrate isolation, future work could involve designing and executing more sophisticated and long-duration attack scenarios and stress tests. This would further validate the trustworthiness and resilience of the SSPFS under diverse and sustained malicious activities.

Bibliography

[1] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. url: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. ii).

[2] T. Alladi et al. "Applications of blockchain in unmanned aerial vehicles: A review". In: *Vehicular Communications* 23 (2020), p. 100249. issn: 2214-2096. doi: https://doi.org/10.1016/j.vehcom.2020.100249. url: https://www.sciencedirect.com/science/article/pii/S2214209620300206 (cit. on pp. 1, 11, 12, 14, 36).

[3] J. Glossner, S. Murphy, and D. Iancu. "An overview of the drone open-source ecosystem". In: *arXiv preprint arXiv:2110.02260* (2021) (cit. on pp. 1, 11, 17, 32).

[4] S. A. H. Mohsan et al. "Towards the unmanned aerial vehicles (UAVs): A comprehensive review". In: *Drones* 6.6 (2022), p. 147 (cit. on pp. 1, 12–15, 18, 19, 35).

[5] B. Nassi et al. "SoK: Security and Privacy in the Age of Commercial Drones". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 1434–1451. doi: 10.1109/SP40001.2021.00005 (cit. on pp. 1, 12, 18, 35).

[6] ISO. *Product development: software level, ISO 26262: Road Vehicles - Functional safety 6* (cit. on pp. 2, 5).

[7] R. (S. 167. *Software considerations in airborne systems and equipment certification*. RTCA, Incorporated, 1992 (cit. on pp. 2, 5).

[8] E. Cenelec. "50128-Railway applications-Communication, signalling and processing systems-Software for railway control and protection systems". In: *Book EN* 50128 (2012) (cit. on p. 2).

[9] A. Burns and R. I. Davis. "Mixed criticality systems-a review:(february 2022)". In: *Department of Computer Science, University of York, Tech. Rep* (2022) (cit. on pp. 2, 5, 6).

[10] M. Cinque et al. “Virtualizing mixed-criticality systems: A survey on industrial trends and issues”. In: *Future Generation Computer Systems* 129 (2022), pp. 315–330. issn: 0167-739X. doi: https://doi.org/10.1016/j.future.2021.12.002. url: https://www.sciencedirect.com/science/article/pii/S0167739X21004787 (cit. on pp. 6–9).

[11] Xilinx. *RunX*. https://github.com/Xilinx/runx. accessed: 2022-11-27. 2020 (cit. on p. 6).

[12] V. Struhár et al. “Real-time containers: A survey”. In: *2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2020 (cit. on p. 6).

[13] S. A.G. *Jailhouse hypervisor source code*. https://github.com/siemens/jailhouse. accessed: 2022-11-30 (cit. on pp. 8, 9).

[14] J. Martins et al. “Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems”. In: *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*. Ed. by M. Bertogna and F. Terraneo. Vol. 77. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020, 3:1–3:14. isbn: 978-3-95977-136-8. doi: 10.4230/OASIcs.NG-RES.2020.3. url: https://drops.dagstuhl.de/opus/volltexte/2020/11779 (cit. on pp. 8–11).

[15] A. Kivity et al. “kvm: the Linux virtual machine monitor”. In: *Proceedings of the Linux symposium*. Vol. 1. 8. Dttawa, Dntorio, Canada. 2007, pp. 225–230 (cit. on pp. 8, 9).

[16] M. Corporation. *Hyper-V*. https://learn.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-technology-overview. accessed: 2022-11-30. 2020 (cit. on p. 8).

[17] P. Barham et al. “Xen and the art of virtualization”. In: *ACM SIGOPS operating systems review* 37.5 (2003), pp. 164–177 (cit. on pp. 8–10).

[18] G. Heiser. “The role of virtualization in embedded systems”. In: *Proceedings of the 1st workshop on Isolation and integration in embedded systems*. 2008, pp. 11–16 (cit. on p. 9).

[19] SysGO. *PikeOS product overview*. https://www.sysgo.com/fileadmin/user_upload/www.sysgo.com/redaktion/downloads/pdf/data-sheets/SYSGO-Product-Overview-PikeOS.pdf. accessed: 2022-11-30 (cit. on p. 9).

[20] M. Masmano et al. “Xtratum: a hypervisor for safety critical embedded systems”. In: *11th Real-Time Linux Workshop*. Citeseer. 2009, pp. 263–272 (cit. on p. 9).

[21] S. Pinto et al. “Towards a TrustZone-assisted hypervisor for real-time embedded systems”. In: *IEEE computer architecture letters* 16.2 (2016), pp. 158–161 (cit. on p. 9).

[22] J. Martins et al. “µTZVisor: A Secure and Safe Real-Time Hypervisor”. In: *Electronics* 6.4 (2017). issn: 2079-9292. doi: 10.3390/electronics6040093. url: https://www.mdpi.com/2079-9292/6/4/93 (cit. on p. 9).

[23] P. Lucas et al. “VOSYSmonitor, a TrustZone-based Hypervisor for ISO 26262 Mixed-critical System”. In: *2018 23rd Conference of Open Innovations Association (FRUCT)*. 2018-11, pp. 231–238. doi: 10.23919/FRUCT.2018.8588018 (cit. on p. 9).

[24] P. Lucas et al. “Vosysmonitor, a low latency monitor layer for mixed-criticality systems on armv8-a”. In: *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017 (cit. on p. 9).

[25] J. Martins et al. “{ClickOS} and the Art of Network Function Virtualization”. In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 2014, pp. 459–473 (cit. on p. 9).

[26] S. Lankes, S. Pickartz, and J. Breitbart. “HermitCore: a unikernel for extreme scale computing”. In: *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*. 2016, pp. 1–8 (cit. on p. 9).

[27] A. Bansal et al. “Evaluating the memory subsystem of a configurable heterogeneous mpsoc”. In: *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*. Vol. 7. 2018, p. 55 (cit. on p. 10).

[28] Q. Ge et al. “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware”. In: *Journal of Cryptographic Engineering* 8.1 (2018), pp. 1–27. doi: 10.1007/s13389-016-0141-6. url: https://doi.org/10.1007/s13389-016-0141-6 (cit. on p. 10).

[29] J. Martins and S. Pinto. “Bao: a modern lightweight embedded hypervisor”. In: *Embedded World Conference*. Nuremberg, Germany. 2020 (cit. on p. 10).

[30] C. Dall. *The Design, Implementation, and Evaluation of Software and Architectural Support for ARM Virtualization*. Columbia University, 2018 (cit. on p. 11).

[31] B. Hypervisor. *Bao project repo*. https://github.com/bao-project/bao-hypervisor. accessed: 2022-11-27. 2020 (cit. on p. 11).

[32] B. G. Blundell. “Empowering Technology: Drones”. In: *Ethics in Computing, Science, and Engineering: A Student’s Guide to Doing Things Right*. Cham: Springer International Publishing, 2020, pp. 489–578. isbn: 978-3-030-27126-8. doi: 10.1007/978-3-030-27126-8_7. url: https://doi.org/10.1007/978-3-030-27126-8_7 (cit. on p. 11).

[33] ConsortIQ. *A not-so-short history of Unmanned Aerial Vehicles (UAV)*. https://consortiq.com/uas-resources/short-history-unmanned-aerial-vehicles-uavs. accessed: 2022-11-30. 2022 (cit. on p. 12).

[34] ArduPilot. *History of ArduPilot*. https://ardupilot.org/copter/docs/common-history-of-ardupilot.html. accessed: 2022-11-30. 2022 (cit. on p. 12).

[35] Auterion. *The story of PX4 and Pixhawk*. https://auterion.com/company/the-history-of-pixhawk/. accessed: 2022-11-30. 2022 (cit. on p. 12).

[36] Z. Ullah, F. Al-Turjman, and L. Mostarda. "Cognition in UAV-Aided 5G and Beyond Communications: A Survey". In: *IEEE Transactions on Cognitive Communications and Networking* 6.3 (2020), pp. 872–891. doi: 10.1109/TCCN.2020.2968311 (cit. on p. 13).

[37] A. Fotouhi et al. "Survey on UAV Cellular Communications: Practical Aspects, Standardization Advancements, Regulation, and Security Challenges". In: *IEEE Communications Surveys & Tutorials* 21.4 (2019-10), pp. 3417–3442. issn: 1553-877X. doi: 10.1109/COMST.2019.2906228 (cit. on p. 13).

[38] C. Stöcker et al. "Review of the Current State of UAV Regulations". In: *Remote Sensing* 9.5 (2017). issn: 2072-4292. doi: 10.3390/rs9050459. url: https://www.mdpi.com/2072-4292/9/5/459 (cit. on pp. 13, 18).

[39] Flyability. *Gas powered drone: A guide*. https://www.flyability.com/gas-powered-drone. accessed: 2022-11-30. 2022 (cit. on p. 14).

[40] Parrot. *Parrot drones*. https://www.parrot.com/us/drones. accessed: 2022-11-30. 2022 (cit. on pp. 14, 15).

[41] Dji. *Dji Mavic 3*. https://www.dji.com/pt/mavic-3. accessed: 2022-11-30. 2022 (cit. on p. 14).

[42] Energyor. *Energyor H2Quad 1000*. http://energyor.com/products/detail/h2quad-1000. accessed: 2022-11-30. 2022 (cit. on pp. 14, 15).

[43] Flaperon. *Flaperon MX8*. https://flaperon.com/. accessed: 2022-11-30. 2022 (cit. on p. 14).

[44] D. Analyst. *The rise of open-source drones*. https://droneanalyst.com/2021/05/30/rise-of-open-source-drones. accessed: 2022-11-30. 2021 (cit. on pp. 14, 23).

[45] Dji. *Dji Software Development Kit*. https://developer.dji.com/. accessed: 2022-11-30. 2022 (cit. on pp. 14, 33).

[46] T. Verge. *Yuneec announces Typhoon H Plus alongise first fixed-wing and racing drones*. https://www.theverge.com/2018/1/9/16867090/yuneec-typhoon-h-plus-firebird-fpv-hd-racer-drones-ces-2018. accessed: 2022-11-30. 2022 (cit. on p. 15).

[47] V. Rotors. *Velos UAV*. https://www.velosuav.com/velosv3/. accessed: 2022-11-30. 2022 (cit. on p. 15).

[48] DeltaQuad. *DeltaQuad Pro VTOL UAV*. https://www.deltaquad.com/. accessed: 2022-11-30. 2022 (cit. on p. 15).

[49] A. A. Company. *Hybrid Advanced Multi-Rotor (HAMR)*. https://advancedaircraftcompany.com/hamr/. accessed: 2022-11-30. 2022 (cit. on p. 15).

[50] XSun. *Solar X One*. https://xsun.fr/autonomous-drone/. accessed: 2022-11-30. 2022 (cit. on p. 15).

[51] S. Aggarwal and N. Kumar. "Path planning techniques for unmanned aerial vehicles: A review, solutions, and challenges". In: *Computer Communications* 149 (2020), pp. 270–299. issn: 0140-3664. doi: https://doi.org/10.1016/j.comcom.2019.10.014. url: https://www.sciencedirect.com/science/article/pii/S0140366419308539 (cit. on p. 16).

[52] T. Vogeltanz. "A Survey of Free Software for the Design, Analysis, Modelling, and Simulation of an Unmanned Aerial Vehicle". In: *Archives of Computational Methods in Engineering* 23.3 (2016), pp. 449–514. doi: 10.1007/s11831-015-9147-y. url: https://doi.org/10.1007/s11831-015-9147-y (cit. on p. 16).

[53] E. Ebeid, M. Skriver, and J. Jin. "A Survey on Open-Source Flight Control Platforms of Unmanned Aerial Vehicle". In: *2017 Euromicro Conference on Digital System Design (DSD)*. 2017, pp. 396–402. doi: 10.1109/DSD.2017.30 (cit. on pp. 17, 22).

[54] D. L. Gabriel, J. Meyer, and F. du Plessis. "Brushless DC motor characterisation and selection for a fixed wing UAV". In: *IEEE Africon '11*. 2011, pp. 1–6. doi: 10.1109/AFRCON.2011.6072087 (cit. on p. 17).

[55] M. Leccadito et al. "A survey on securing UAS cyber physical systems". In: *IEEE Aerospace and Electronic Systems Magazine* 33.10 (2018), pp. 22–32 (cit. on pp. 18, 20, 29).

[56] C. G. L. Krishna and R. R. Murphy. "A review on cybersecurity vulnerabilities for unmanned aerial vehicles". In: *2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. 2017, pp. 194–199. doi: 10.1109/SSRR.2017.8088163 (cit. on pp. 18, 35).

[57] K. Mansfield et al. "Unmanned aerial vehicle smart device ground control station cyber security threat model". In: *2013 IEEE International Conference on Technologies for Homeland Security (HST)*. 2013, pp. 722–728. doi: 10.1109/THS.2013.6699093 (cit. on p. 18).

[58] sUAS News. *US Army calls for units to discontinue use of DJI equipment*. https://www.suasnews.com/2017/08/us-army-calls-units-discontinue-use-dji-equipment/. accessed: 2022-12-05. 2017 (cit. on pp. 18, 23).

[59] T. Independent. *Man arrested for landing 'radioactive' drone on japanese prime minister's roof*. https://www.independent.co.uk/news/world/asia/man-arrested-for-landing-radioactive-drone-on-japanese-prime-ministers-roof-10203517.html. accessed: 2022-12-05. 2015 (cit. on p. 18).

[60] N. Times. *Venezuelan President Target By Drone Attack, Officials say*. https://www.nytimes.com/2018/08/04/world/americas/venezuelan-president-targeted-in-attack-attempt-minister-says.html. accessed: 2022-12-05. 2018 (cit. on p. 18).

[61] T. Drive. *Russia Offers New Details About Syrian Mass Drone Attack, Now Implies Ukrainian Connection*. https://www.thedrive.com/the-war-zone/17595/russia-offers-new-details-about-syrian-mass-drone-attack-now-implies-ukrainian-connection. accessed: 2022-12-05. 2019 (cit. on p. 18).

[62] D. Sathyamoorthy. "A review of security threats of unmanned aerial vehicles and mitigation steps". In: *J. Def. Secur* 6.1 (2015), pp. 81–97 (cit. on p. 19).

[63] I. G. Ferrão et al. "STUART: ReSilient archiTecture to dynamically manage Unmanned aeriAl vehicle networks undeR atTack". In: *2020 IEEE Symposium on Computers and Communications (ISCC)*. IEEE. 2020, pp. 1–6 (cit. on pp. 19, 20, 35).

[64] P. Autopilot. *PX4 System Architecture*. https://docs.px4.io/main/en/concept/px4_systems_architecture.html. accessed: 2022-12-08. 2022 (cit. on pp. 20, 29, 31).

[65] F. S. Foundation. *GNU Operating System*. https://gnu.org/philosophy/free-hardware-designs.en.html. accessed: 2022-12-05. 2022 (cit. on pp. 22, 31).

[66] P. Autopilot. *Pixhawk 4*. https://docs.px4.io/main/en/flight_controller/pixhawk4.html. accessed: 2022-12-07. 2022 (cit. on pp. 22, 23).

[67] Paparrazi. *Paparazzi Chimera v1.00*. https://wiki.paparazziuav.org/wiki/Chimera/v1.00. accessed: 2022-12-07. 2022 (cit. on pp. 22, 23).

[68] O. Wiki. *Copter Control / CC3D / Atom Hardware Setup*. https://opwiki.readthedocs.io/en/latest/user_manual/cc3d/cc3d.html. accessed: 2022-12-07. 2022 (cit. on pp. 22, 24).

[69] A. Wiki. *CUAV v5 Plus Overview*. https://ardupilot.org/copter/docs/common-cuav-v5plus-overview.html. accessed: 2022-12-07. 2022 (cit. on pp. 22, 24).

[70] DroneDJ. *After product ban, the US DoD formally blacklists drone giant DJI (Update)*. https://dronedj.com/2022/10/07/dji-dod-drone/. accessed: 2022-12-08. 2022 (cit. on p. 23).

[71] A. Wiki. *SPRacing H7 extreme*. https://ardupilot.org/copter/docs/common-spracingh7-extreme.html. accessed: 2022-12-08. 2022 (cit. on pp. 24, 25).

[72] Dronecode. *Aerotenna OcPoC-Zynq Mini Flight Controller*. https://docs.px4.io/v1.9.0/en/flight_controller/ocpoc_zynq.html. accessed: 2022-12-08. 2022 (cit. on p. 25).

[73] A. Wiki. *Navio2 Overview*. https://ardupilot.org/copter/docs/common-navio2-overview.html. accessed: 2022-12-07. 2022 (cit. on pp. 25, 26).

[74] A. Wiki. *Horizon31 PixC4-Jetson*. https://ardupilot.org/copter/docs/common-horizon31-pixc4-jetson.html. accessed: 2022-12-08. 2022 (cit. on pp. 25–27).

[75] Auterion. *Skynode X*. https://auterion.com/product/skynode-x/. accessed: 2024-07-30. 2024 (cit. on p. 26).

[76] Auterion. *Skynode X Datasheet*. https://3329189600-files.gitbook.io/~/files/v0/b/gitbook-x-prod.appspot.com/o/spaces%2FFW1Ge1p1f6WHyiYCb146%2Fuploads%2Fg5ziQAabZrGnMfLuX7OC%2FSkynode%20X%20Datasheet.pdf?alt=media&token=4c36ecb2-4299-4551-afa5-2a1d8c83841a. accessed: 2024-07-30. 2024 (cit. on p. 28).

[77] LucidBotsShop. *Auterion Flight Controller Sky Node - Enterprise Edition*. https://lucidbots.shop/products/auterion-flight-conroller-sky-node. accessed: 2024-07-30. 2024 (cit. on p. 28).

[78] Auterion. *Auterion Announces New All-In-One solution for Small Unmanned Systems*. https://auterion.com/auterion-announces-new-all-in-one-solution-for-small-unmanned-systems/. accessed: 2024-07-30. 2024 (cit. on p. 28).

[79] BreakingDefense. *Auterion Announces New All-In-One solution for Small Unmanned Systems*. https://breakingdefense.com/2024/06/skynode-s-auterion-autonomy-kit-lets-attack-drones-fly-through-jamming/. accessed: 2024-08-05. 2024 (cit. on p. 29).

[80] DefenseExpress. *Ukraine Receives Skynode S Universal Machine Vision for Drones from American Company Auterion*. https://en.defence-ua.com/weapon_and_tech/ukraine_receives_skynode_s_universal_machine_vision_for_drones_from_american_company_auterion-11011.html. accessed: 2024-08-05. 2024 (cit. on p. 29).

[81] P. Github. *PX4-Autopilot*. https://github.com/PX4/PX4-Autopilot. accessed: 2022-12-08. 2022 (cit. on p. 31).

[82] P. Autopilot. *Opens Source Autopilot for Drone Developers*. https://px4.io/. accessed: 2022-12-08. 2022 (cit. on p. 31).

[83] T. Jargalsaikhan et al. "Architectural Process for Flight Control Software of Unmanned Aerial Vehicle with Module-Level Portability". In: *Aerospace* 9.2 (2022), p. 62 (cit. on pp. 31–33).

[84] A. Github. *ArduPilot Autopilot*. https://github.com/ArduPilot/ardupilot. accessed: 2022-12-08. 2022 (cit. on p. 32).

[85] ArduPilot. *ArduPilot, Home*. https://ardupilot.org/. accessed: 2022-12-08. 2022 (cit. on p. 32).

[86] Paparazzi. *Paparazzi Home*. https://wiki.paparazziuav.org/wiki/Main_Page. accessed: 2022-12-08. 2022 (cit. on p. 32).

[87] P. U. Github. *Paparazzi UAS Autopilot*. https://github.com/paparazzi/paparazzi. accessed: 2022-12-08. 2022 (cit. on p. 32).

[88] PX4. *Auterion Skynode*. https://docs.px4.io/main/en/companion_computer/auterion_skynode.html. accessed: 2024-08-06. 2024 (cit. on p. 33).

[89] Auterion. *Structuring Applications with Multiple Services*. https://docs.auterion.com/app-development/app-framework/structuring-applications-with-multiple-services. accessed: 2024-08-07. 2024 (cit. on pp. 33, 36).

[90] Auterion. *Auterion Mission Control*. https://auterion.com/product/mission-control/. accessed: 2024-08-06. 2024 (cit. on p. 33).

[91] Auterion. *Auterion Suite*. https://auterion.com/product/suite/. accessed: 2024-08-06. 2024 (cit. on p. 33).

[92] W. Blog. *Northrop Grumman X-47B UCAS-D Running on Wind River VxWorks Catapults from Aircraft Carrier*. https://blogs.windriver.com/wind_river_blog/2013/05/historic-milestone-for-northrops-x-47b-and-wind-river/. accessed: 2022-12-09. 2013 (cit. on p. 34).

[93] A. Internation. *Wind River VxWorks 653 Providing Power for Airbus Helionix*. https://www.aviationtoday.com/2016/05/11/wind-river-vxworks-653-providing-power-for-airbus-helionix/. accessed: 2022-12-09. 2016 (cit. on p. 34).

[94] sUAS News. *Wind River Technology Powers Airbus Group's Innovative Unmanned Aerial Vehicle ATLANTE*. https://www.suasnews.com/2014/09/wind-river-technology-powers-airbus-groups-innovative-unmanned-aerial-vehicle-atlante/. accessed: 2022-12-09. 2014 (cit. on p. 34).

[95] M. Chong and L. Chuntao. "Design of flight control software for small unmanned aerial vehicle based on VxWorks". In: *Proceedings of 2014 IEEE Chinese Guidance, Navigation and Control Conference*. 2014, pp. 1831–1834. doi: 10.1109/CGNCC.2014.7007459 (cit. on pp. 34, 35).

[96] K. K. G. Buquerin. "Security Evaluation for the Real-time Operating System VxWorks 7 for Avionic Systems". PhD thesis. Technische Hochschule Ingolstadt, 2018 (cit. on p. 35).

[97] M. Zhang et al. "Which Is the Best Real-Time Operating System for Drones? Evaluation of the Real-Time Characteristics of NuttX and ChibiOS". In: *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE. 2021, pp. 582–590 (cit. on p. 36).

[98] T. Fautrel et al. "An hypervisor approach for mixed critical real-time UAV applications". In: *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE. 2019, pp. 985–991 (cit. on p. 36).

[99] O. Sami Oubbati et al. “Softwarization of UAV Networks: A Survey of Applications and Future Trends”. In: *IEEE Access* 8 (2020), pp. 98073–98125. doi: 10.1109/ACCESS.2020.2994494 (cit. on p. 37).

[100] B. Nogales et al. “Adaptable and Automated Small UAV Deployments via Virtualization”. In: *Sensors* 18.12 (2018). issn: 1424-8220. doi: 10.3390/s18124116. url: https://www.mdpi.com/1424-8220/18/12/4116 (cit. on p. 37).

[101] P. Autopilot. *PX4 Companion Computer: Routers*. https://docs.px4.io/main/en/companion_computer/#routers. accessed: 2024-10-08. 2022 (cit. on p. 42).

[102] P. Autopilot. *PX4 MAVLink Cameras: Camera managers*. https://docs.px4.io/main/en/camera/mavlink_v2_camera.html#camera-managers. accessed: 2024-10-08. 2022 (cit. on p. 42).

[103] N. Semiconductors. *NXP HoverGames drone kit including RDDRONE-FMUK66 and peripherals*. https://www.nxp.com/design/design-center/development-boards-and-designs/nxp-hovergames-drone-kit-including-rddrone-fmuk66-and-peripherals:KIT-HGDRONEK66. accessed: 2024-06-08 (cit. on pp. 46, 47).

[104] N. Semiconductors. *PX4 Robotic Drone Vehicle/Flight Management Unit (VMU/FMU) - RDDRONE-FMUK66*. https://www.nxp.com/design/design-center/development-boards-and-designs/px4-robotic-drone-vehicle-flight-management-unit-vmu-fmu-rddrone-fmuk66:RDDRONE-FMUK66. accessed: 2024-06-08 (cit. on p. 46).

[105] PX4. *Experimental Autopilots*. https://docs.px4.io/main/en/flight_controller/autopilot_experimental.html. accessed: 2024-06-12 (cit. on p. 47).

[106] PX4. *RPi PilotPi Shield*. https://docs.px4.io/main/en/flight_controller/raspberry_pi_pilotpi.html. accessed: 2024-06-13 (cit. on pp. 47, 48).

[107] R. Pi. *Raspberry Pi 4 Model B specifications*. https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/. accessed: 2024-06-14 (cit. on p. 48).

[108] R. Pi. *Processors*. https://www.raspberrypi.com/documentation/computers/processors.html. accessed: 2024-06-14 (cit. on pp. 48, 75).

[109] R. P. F. Github. *Raspberry Pi Firmware Wiki*. https://github.com/raspberrypi/firmware/wiki. accessed: 2024-06-27 (cit. on p. 49).

[110] R. P. L. kernel Github. *Mailbox device driver*. https://github.com/raspberrypi/linux/blob/rpi-6.6.y/drivers/firmware/raspberrypi.c. accessed: 2024-07-30 (cit. on p. 51).

[111] R. P. Ltd. *BCM2711 ARM Peripherals*. Raspberry Pi Ltd, 2022, p. 6 (cit. on p. 52).

[112] Creative. *Creative Live! Cam Sync 1080p V2*. https://en.creative.com/p/webcams/creative-live-cam-sync-1080p-v2. accessed: 2024-08-10 (cit. on p. 52).

[113] Amazon. *EDUP AX3000M USB 6E WiFi Adapter for PC*. https://www.amazon.es/-/en/dp/B0DFYBZSR6. accessed: 2024-08-10 (cit. on p. 52).

[114] Morrownr. *USB WiFi Adapters that are supported with Linux in-kernel drivers*. https://github.com/morrownr/USB-WiFi/blob/main/home/USB_WiFi_Adapters_that_are_supported_with_Linux_in-kernel_drivers.md#axe3000---usb30---24-ghz-5-ghz-and-6-ghz-wifi-6e. accessed: 2024-08-10 (cit. on p. 52).

[115] u-blox. *NEO-M8 series*. https://www.u-blox.com/sites/default/files/products/documents/NEO-M8_ProductSummary_UBX-16000345.pdf. accessed: 2024-07-10 (cit. on p. 60).

[116] R. Modelismo. *LiPo Stick Pack 11.1V-50C 5000 Hardcase (XT60)*. https://rtr-modelismo.com/pt/electronica-baterias-li-po/107050-lipo-stick-pack-11-1v-50c-5000-hardcase-4250650938475.html?gad_source=1. accessed: 2024-07-10 (cit. on p. 60).

[117] B. E. C. Referencer. *panic.c*. https://elixir.bootlin.com/linux/v6.6.53/source/kernel/panic.c. accessed: 2025-01-27. 2024 (cit. on p. 73).

[118] Bootlin-Elixir-Cross-Referencer. *iomap.c*. https://elixir.bootlin.com/linux/v6.6.53/source/lib/iomap.c#L221. accessed: 2025-01-28. 2024 (cit. on p. 75).

[119] J. Martins and S. Pinto. "Shedding light on static partitioning hypervisors for arm-based mixed-criticality systems". In: *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2023, pp. 40–53 (cit. on p. 78).

[120] M. R. Guthaus et al. "MiBench: A free, commercially representative embedded benchmark suite". In: *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE. 2001, pp. 3–14 (cit. on p. 80).

[121] PerfWiki. *perf: Linux profiling with performance counters*. https://perfwiki.github.io/main/. accessed: 2025-01-28. 2024 (cit. on p. 80).

[122] J. Martins and S. Pinto. *Shedding Light repository*. https://github.com/ESRGv3/shedding-light-static-partitioning-hypervisors. accessed: 2025-01-27. 2022 (cit. on p. 80).

[123] PX4. *Module Template for Full applications – Workqueue*. https://docs.px4.io/main/en/modules/module_template.html. accessed: 2025-01-30. 2024 (cit. on p. 82).

[124] PX4. *Modules Reference – Controller*. https://docs.px4.io/main/en/modules/modules_controller.html. accessed: 2025-01-30. 2024 (cit. on p. 82).

[125] PX4. *Modules Reference – Drivers*. https://docs.px4.io/main/en/modules/modules_driver.html. accessed: 2025-01-30. 2024 (cit. on p. 82).

[126] PX4. *Modules Reference – System*. https://docs.px4.io/main/en/modules/modules_system.html. accessed: 2025-01-30. 2024 (cit. on p. 82).

[127] PX4. *Mission mode (Multicopter) – Rounded turns: Inter-Waypoint Trajectory*. https://docs.px4.io/main/en/flight_modes_mc/mission.html#rounded-turns-inter-waypoint-trajectory. accessed: 2025-06-28. 2024 (cit. on p. 88).

A

Taxonomy

The following appendix contains the taxonomies for the virtualization and UAV concepts.

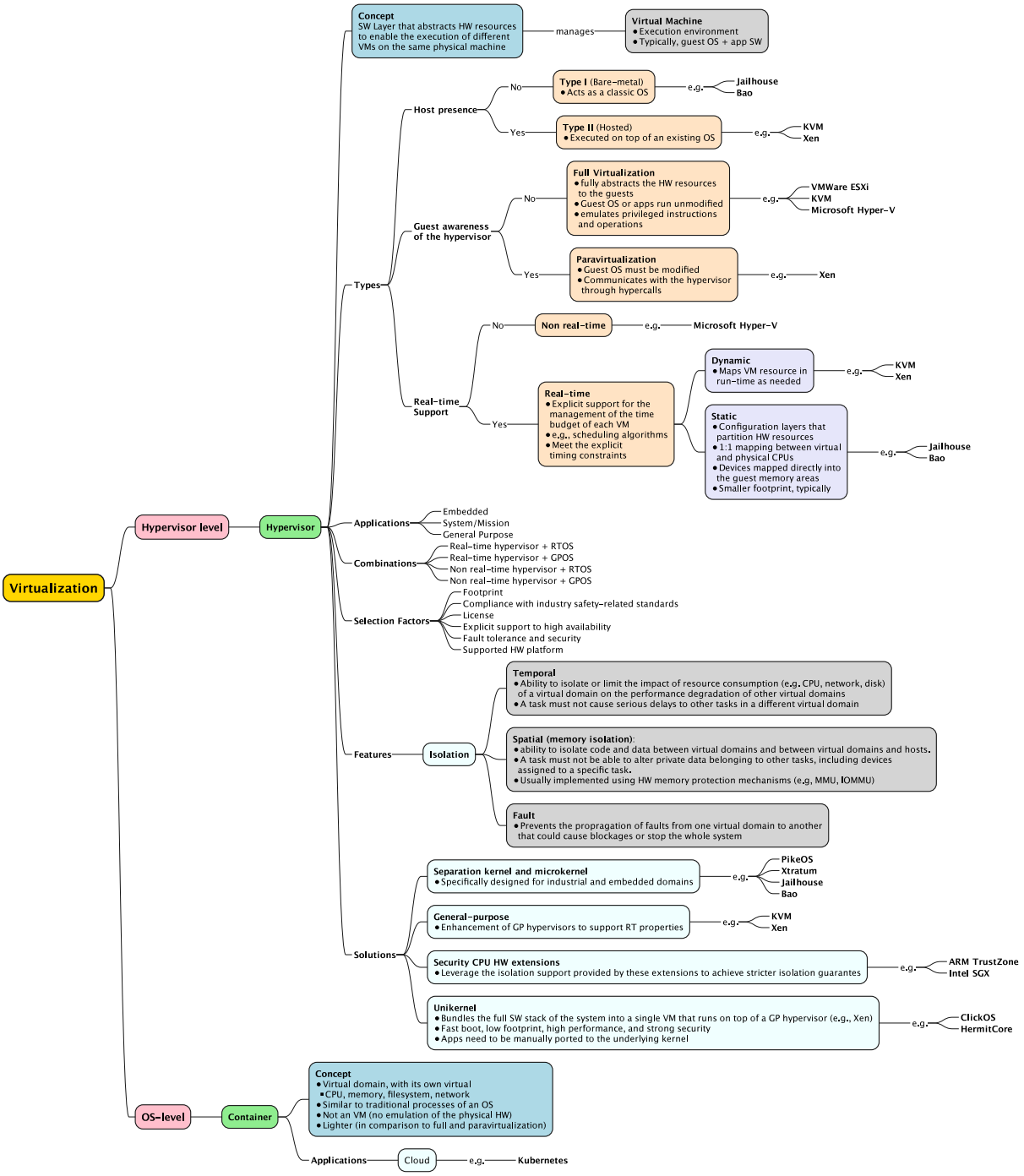


Figure 75: Virtualization mind map

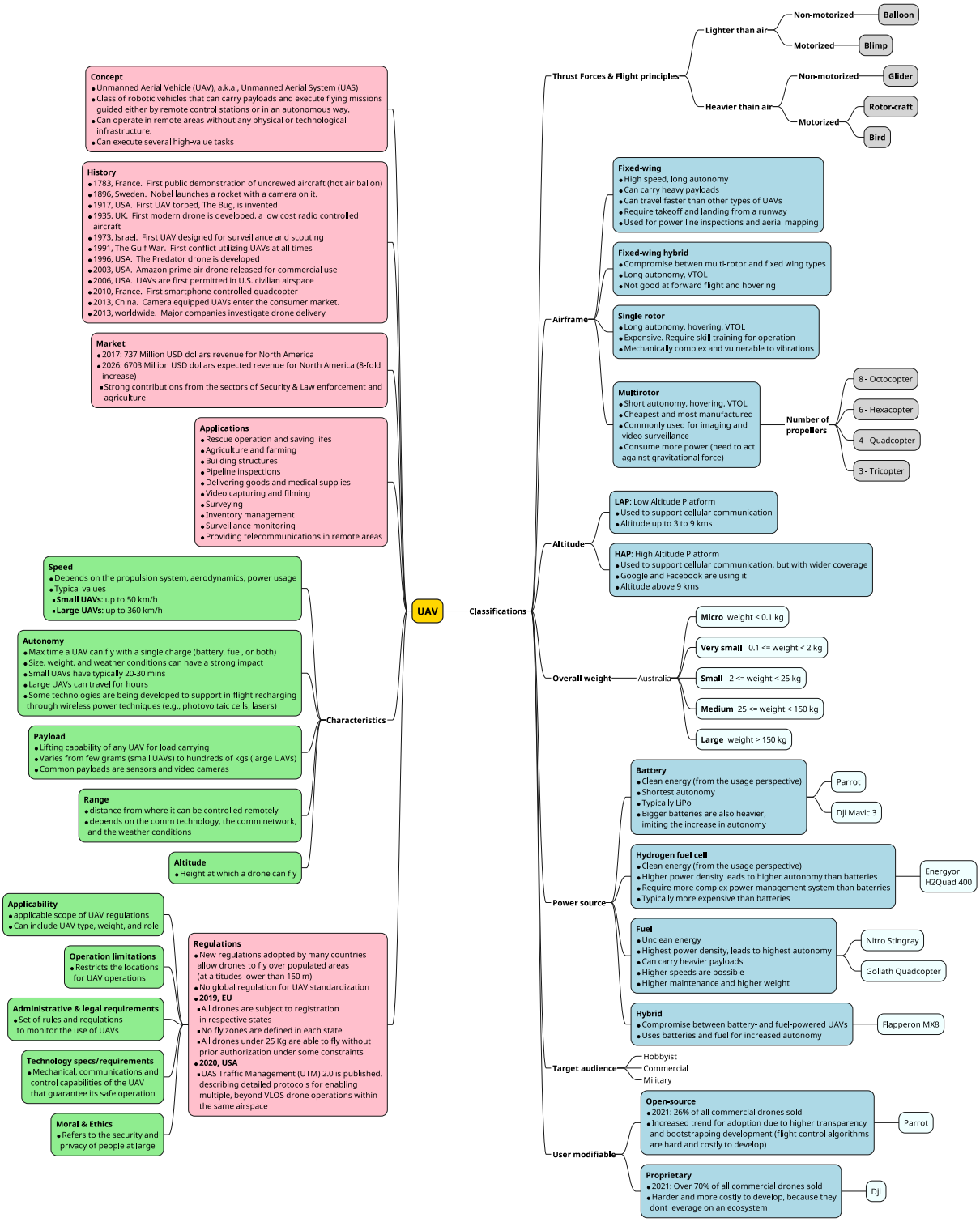


Figure 76: UAV mind map: generic overview

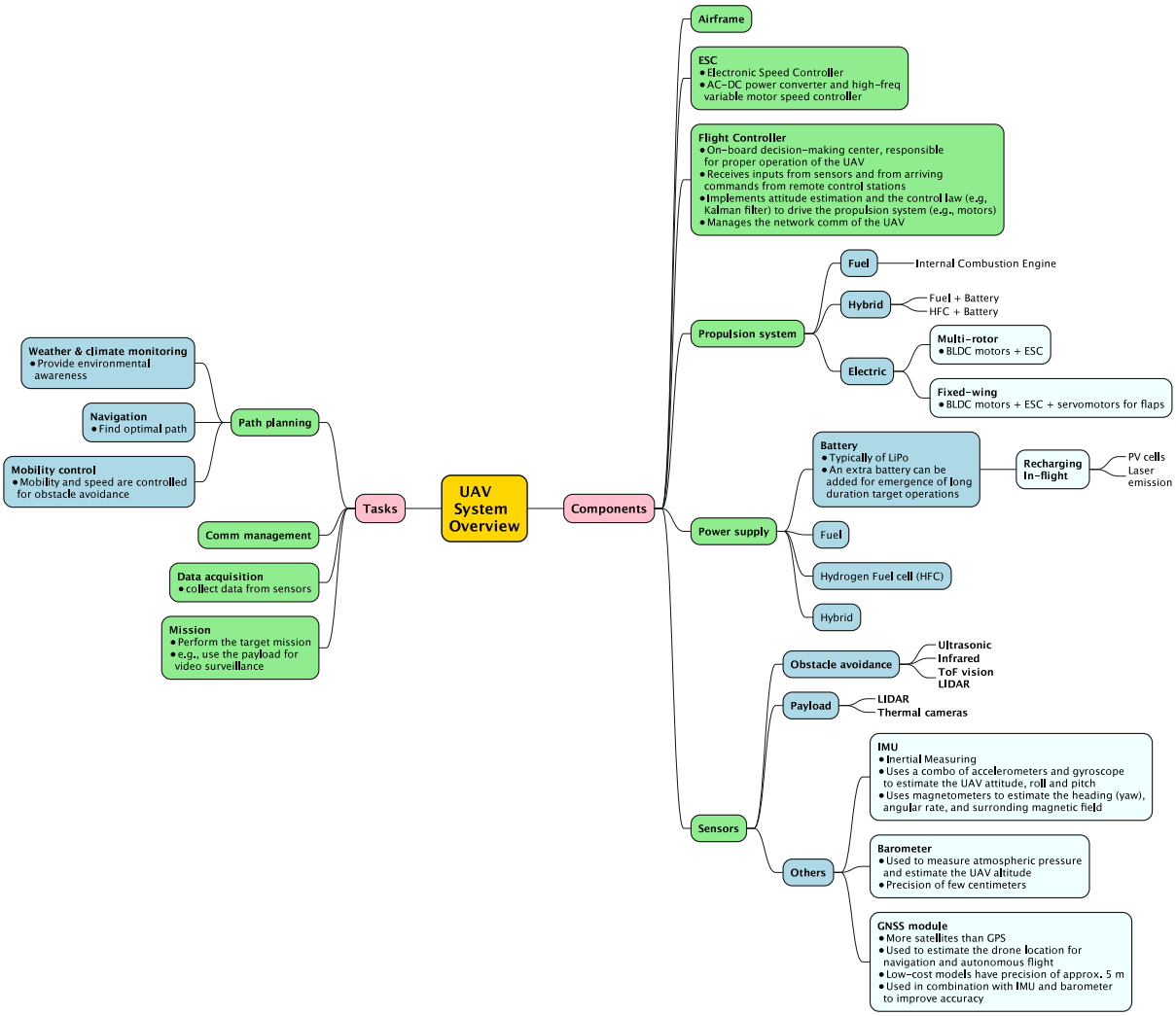


Figure 77: UAV mind map: System overview — Tasks and components

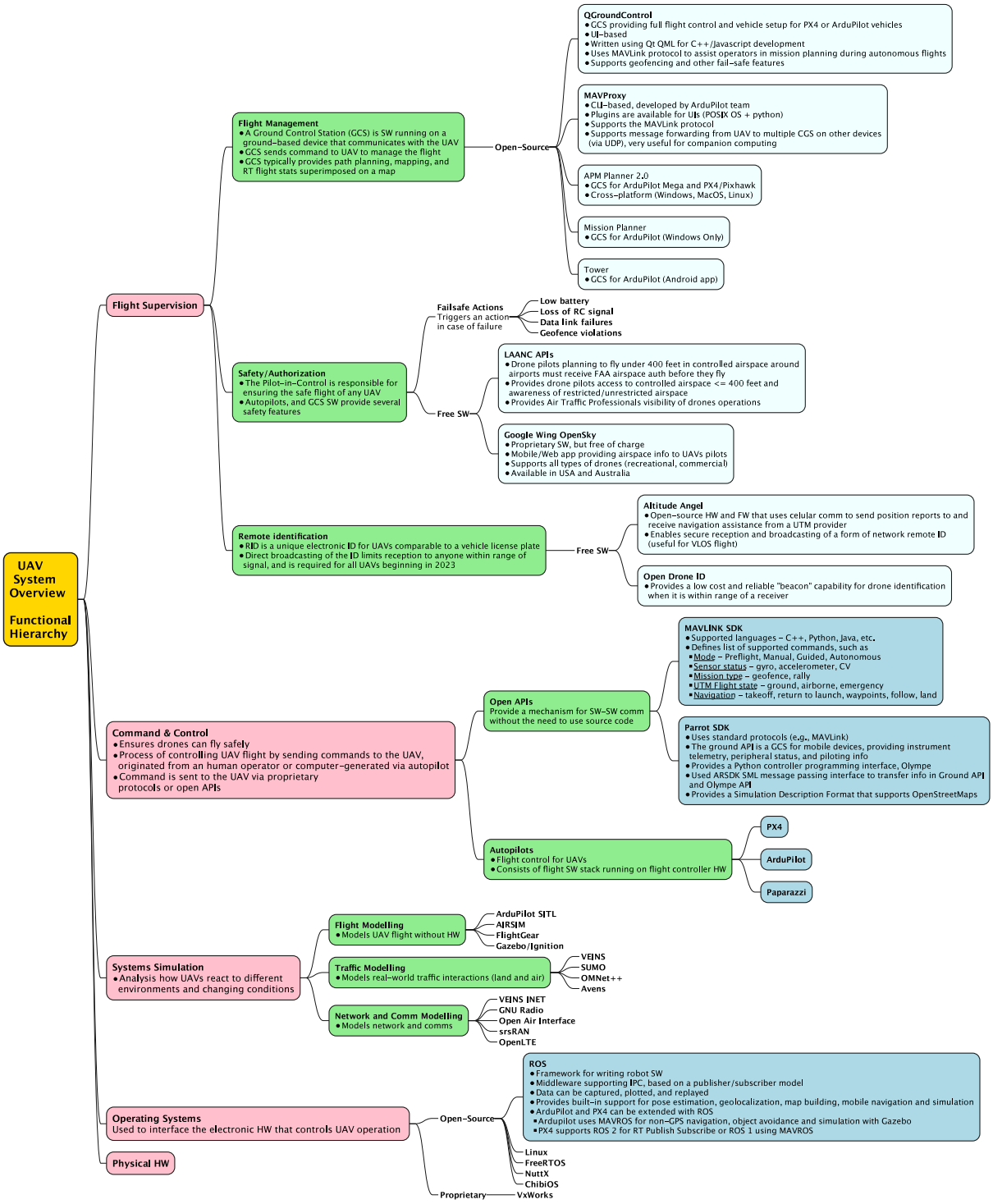


Figure 78: UAV mind map: System overview — Functional Hierarchy

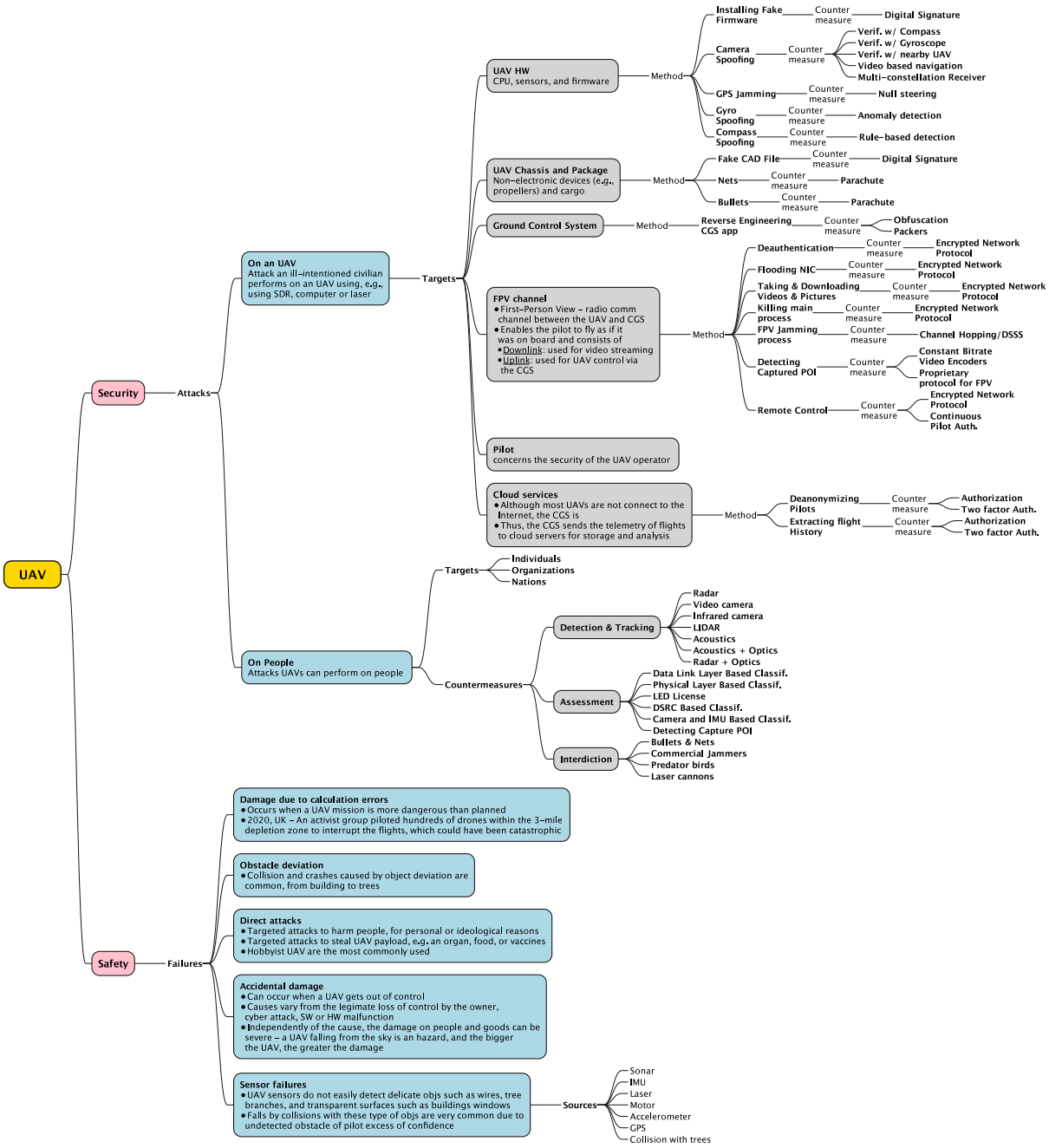


Figure 79: UAV mind map: security and safety

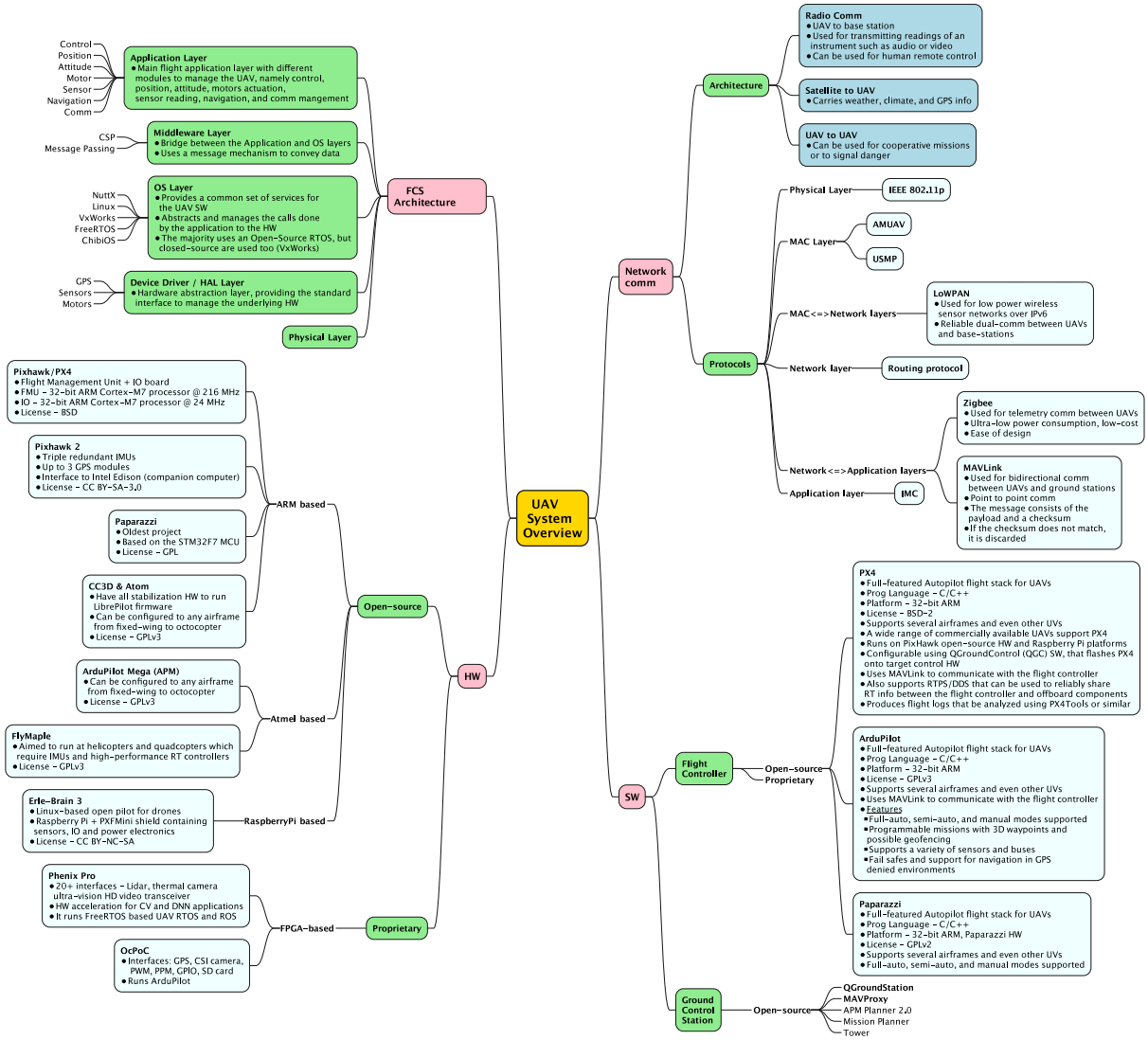


Figure 80: UAV mind map: System overview — Architecture, HW, SW and communications

B

Source code listings

The following appendix presents the source code listings used in the implementation (Chapter 4) and evaluation (Chapter 5) of the present work.

```
#!/bin/sh

PX4_DIR=Autopilot
PX4_BRANCH=v1.14.0
PX4_VENV=px4-venv

# Download Autopilot
git clone https://github.com/PX4/PX4-Autopilot.git "$PX4_DIR" \
    --branch "$PX4_BRANCH" --recursive
git submodule sync --recursive
git submodule update --recursive
# Setup virtual environment for the build
python3 -m venv "$PX4_VENV"
source "$PX4_VENV/bin/activate"
# Build PX4 for PilotPi
make -C "$PX4_DIR" scumaker_pilotpi_arm64
# Upload PX4 to PilotPi
export autopilot_host=192.168.1.55
export autopilot_user=pilotpi
make -C "$PX4_DIR" scumaker_pilotpi_arm64 upload # to host
make -C "$PX4_DIR" scumaker_pilotpi_arm64 upload_local_br # to local buildroot
```

Listing B.1: PX4 build script

```
#!/bin/sh
gst-launch-1.0 -v v4l2src device=/dev/video2 ! image/jpeg,width=640,height=480,framerate=30/1 ! jpegdec !
↪  videoconvert ! x264enc bitrate=10000 tune=zerolatency ! rtph264pay config-interval=10 pt=96 ! udpsink
↪  host=192.168.1.37 port=5000
```

Listing B.2: Video surveillance sender script

```
#!/bin/sh
gst-launch-1.0 -v udpsrc port=5000 caps="application/x-rtp,media=video,encoding-name=H264,payload=96" !
↪  queue ! rtph264depay ! queue ! avdec_h264 ! queue ! videoconvert ! queue ! autovideosink sync=false
```

Listing B.3: Video surveillance receiver script

```
define build-atf
    @echo 'ATF: Fix UART5 PL011 for PilotPi...'
    sed -i 's/serial0/serial5/g' $(atf_src)/plat/rpi/rpi4/rpi4_bl31_setup.c
    sed -i 's/#define RPI4_IO_PL011_UART_OFFSET\tULL(0x00201000)/#define
    ↪  RPI4_IO_PL011_UART_OFFSET\tULL(0x00201a00)/' $(atf_src)/plat/rpi/rpi4/include/rpi_hw.h
    @echo 'ATF: Building...'
    $(MAKE) -C $(atf_src) bl31 PLAT=rpi4
endef
```

Listing B.4: USPFS: ATF patch

```
uboot_repo:=https://github.com/u-boot/u-boot.git
uboot_version:=v2024.07
uboot_src:=$(wrkdir_src)/u-boot
uboot_load_bin:=linux.bin
uboot_load_bin_addr:=0x80000
env_file:=$(uboot_src)/board/raspberrypi/rpi/rpi.env
uboot_defconfig:=rpi_4_defconfig
uboot_image:=$(wrkdir_plat_imgs)/u-boot.bin

$(uboot_src):
    @echo 'U-BOOT: Downloading...'
    git clone --depth 1 --branch $(uboot_version) $(uboot_repo) $(uboot_src)

define build-uboot
$(strip $1): $(uboot_src)
    @echo 'U-BOOT: Configuring...'
    $(MAKE) -C $(uboot_src) $(strip $2)
    @echo 'U-BOOT: Modifying environment...'
    @printf "\n\nbootcmd_fatload=fatload mmc 0 $(uboot_load_bin_addr) $(uboot_load_bin); go
    ↪  $(uboot_load_bin_addr)\n" >> $(env_file)
    @printf "bootcmd=run bootcmd_fatload\n" >> $(env_file)
    @echo 'U-BOOT: Building...'
    $(MAKE) -C $(uboot_src) -j$(nproc)
endef

$(eval $(call build-uboot, $(uboot_image), $(uboot_defconfig)))
```

Listing B.5: USPFS: U-Boot makefile

```
struct platform platform = {
    .cpu_num = 4,
    .region_num = (RPI4_MEM_GB > 1) ? 2: 1,
    .regions =  (struct mem_region[]) {
        {
            .base = 0x80000,
            .size = 0x40000000 - 0x80000 - 0x4c00000,
        },
        {
            .base = 0x40000000,
            .size = ((RPI4_MEM_GB-1) * 0x40000000ULL) - 0x4000000,
```

```
        },
    },
    .console = {
        .base = 0xfe201000, /**< Page aligned address; the offset is added on platform.h */
    },
    .arch = {
        .gic = {
            .gicd_addr = 0xff841000,
            .gicc_addr = 0xff842000,
            .gich_addr = 0xff844000,
            .gicv_addr = 0xff846000,
            .maintenance_id = 25,
        },
    },
};
```

Listing B.6: SSPFS: Bao's Raspberry Pi 4 platform description patch

```
#define UART_CLK 48000000 /**< PL011 runs at 48 MHz */
#define PL011_PAGE_OFFSET (0xa00) /**< UART5 offset for page alignment */

#include <drivers/pl011_uart.h>
```

Listing B.7: SSPFS: Bao's Raspberry Pi 4 platform header patch

```
#define RPI_MAILBOX_IRQ_ID 65
#define RPI_HYP_ARG_START  1
#define RPI_HYP_ARG_END    2

spinlock_t rpi_firmware_lock = SPINLOCK_INITVAL;

static void rpi_mailbox_irq_handler(irqid_t irq_id) {
  vcpu_inject_irq(cpu()->vcpu, irq_id);
  interrupts_cpu_enable(irq_id, false);
}

void plat_rpi_init(void) {
  interrupts_reserve(RPI_MAILBOX_IRQ_ID, rpi_mailbox_irq_handler);
}

long rpi_mailbox_hypercall(unsigned long arg0, unsigned long arg1, unsigned long arg2)
{
  switch (arg0) {
  case RPI_HYP_ARG_START:
    spin_lock(&rpi_firmware_lock);
    interrupts_cpu_enable(RPI_MAILBOX_IRQ_ID, true);
    break;

  case RPI_HYP_ARG_END:
    interrupts_cpu_enable(RPI_MAILBOX_IRQ_ID, false);
    spin_unlock(&rpi_firmware_lock);
    break;

  default:
    ERROR("func %s, unknown arg0 = %lu", __func__, arg0);
```

```
  }

  return 0;
}
```

Listing B.8: SSPFS: Mailbox manager added to Bao

```
void init(cpuid_t cpu_id, paddr_t load_addr) {
  /**< Initialize CPUs, memory, and console */
  cpu_init(cpu_id, load_addr);
  mem_init(load_addr);
  console_init();

  if (cpu_is_master())
    console_printk("Bao Hypervisor\n\r");

  interrupts_init(); /**< Initialize interrupt manager */

  if (cpu_is_master())
    plat_rpi_init(); /**< Initialize RPi platform */

  vmm_init(); /**< Initialize VM Manager */

  /* Should never reach here */
  while (1) { }
}
```

Listing B.9: SSPFS: Bao initialization – Raspberry Pi platform initialization

```
enum { HC_INVAL = 0, HC_IPC = 1, HC_RPI_FIRMWARE = 2 };

long int hypercall(unsigned long id)
{
  long int ret = -HC_E_INVAL_ID;

  unsigned long ipc_id = vcpu_readreg(cpu()->vcpu, HYPCALL_ARG_REG(0));
  unsigned long arg1 = vcpu_readreg(cpu()->vcpu, HYPCALL_ARG_REG(1));
  unsigned long arg2 = vcpu_readreg(cpu()->vcpu, HYPCALL_ARG_REG(2));

  switch (id) {
  case HC_IPC:
    ret = ipc_hypercall(ipc_id, arg1, arg2);
    break;
  case HC_RPI_FIRMWARE:
    ret = rpi_mailbox_hypercall(ipc_id, arg1, arg2);
    break;
  default:
    WARNING("Unknown hypercall id %lu", id);
  }

  return ret;
}
```

Listing B.10: SSPFS: Bao hypercall manager – Raspberry Pi firmware mailbox handling

```
#define RPI_HYP_ARG_START  1
#define RPI_HYP_ARG_END 2
enum {HC_INVAL = 0, HC_IPC = 1, HC_RPI_FIRMWARE = 2};

static int
rpi_firmware_transaction(struct rpi_firmware *fw, u32 chan, u32 data)
{
    u32 message = MBOX_MSG(chan, data);
    int ret;

    WARN_ON(data & 0xf);

    mutex_lock(&transaction_lock);

    /* HYP call to lock mailbox access */
    register uint64_t x0 asm("x0") =
        ARM_SMCCC_CALL_VAL(ARM_SMCCC_FAST_CALL, ARM_SMCCC_SMC_64, ARM_SMCCC_OWNER_VENDOR_HYP,
        ↪  HC_RPI_FIRMWARE);
    register uint64_t aux = x0;
    register uint64_t x1 asm("x1") = RPI_HYP_ARG_START;
    register uint64_t x2 asm("x2") = 0; /* DO NOT CARE */
    asm volatile("hvc 0" : "=r"(x0) : "r"(x0), "r"(x1), "r"(x2) : "memory", "cc");

    /* Check for errors */
    if ((int64_t)x0 < 0) {
      dev_err(fw->cl.dev, "HYPCALL START failed: ret = 0x%llx\n", x0);
      return -EINVAL; /* Abort */
    }

    reinit_completion(&fw->c);
    ret = mbox_send_message(fw->chan, &message);
    if (ret >= 0) {
        if (wait_for_completion_timeout(&fw->c, HZ)) {
            ret = 0;
        } else {
            ret = -ETIMEDOUT;
            WARN_ONCE(1, "Firmware transaction timeout");
        }
    } else {
        dev_err(fw->cl.dev, "mbox_send_message failed for chan: %u, data: 0x%08x, ret: %d\n", chan, data,
        ↪  ret);
    }

    /* HYP call to unlock mailbox access */
    x0 = ARM_SMCCC_CALL_VAL(ARM_SMCCC_FAST_CALL, ARM_SMCCC_SMC_64,
                ARM_SMCCC_OWNER_VENDOR_HYP, HC_RPI_FIRMWARE);
    aux = x0;
    x1 = RPI_HYP_ARG_END;
    x2 = 0; /* DO NOT CARE */
    asm volatile("hvc 0" : "=r"(x0) : "r"(x0), "r"(x1), "r"(x2) : "memory", "cc");

    /* Check for errors */
    if ((int64_t)x0 < 0) {
      dev_err(fw->cl.dev, "HYPCALL END failed: ret = 0x%llx\n", x0);
      return -EINVAL; /* Abort */
    }

```

```
    mutex_unlock(&transaction_lock);

    return ret;
}
```

Listing B.11: SSPFS: Linux's Raspberry Pi mailbox driver – patch

```
#include <linux/module.h>
#include <linux/kernel.h>

static int __init panic_init(void) {
    panic("Forced kernel panic!");
    return 0;
}

static void __exit panic_exit(void){
  return;
}

module_init(panic_init);
module_exit(panic_exit);
```

Listing B.12: Functional tests: implementation of the Panic kernel module

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/ioport.h>
#include <linux/io.h>

static u64 reserved_phys_addr = 0;     // Set via module parameter
#define RESERVED_SIZE 0x100000         // 1MB

module_param_named(reserved_phys_addr, reserved_phys_addr, ullong, 0);
MODULE_PARM_DESC(reserved_phys_addr, "Physical address of reserved memory region (hex)");

static int __init memcorrupt_init(void)
{
    void __iomem *addr;
    if (reserved_phys_addr == 0) {
        pr_err("[MEM CORRUPT]: reserved_phys_addr not specified!\n");
        return -EINVAL;
    }

    // Map the physical address to kernel virtual address
    addr = ioremap(reserved_phys_addr, RESERVED_SIZE);
    if (!addr) {
        pr_err("[MEM CORRUPT]: Failed to ioremap 0x%llx\n", reserved_phys_addr);
        release_mem_region(reserved_phys_addr, RESERVED_SIZE);
        return -ENOMEM;
    }

    // Write to the reserved memory to trigger a crash
    pr_info("[MEM CORRUPT]: Writing to reserved memory at %px (physical 0x%llx)\n", addr,
    ↪  reserved_phys_addr);
    iowrite32(0xdeadbeef, addr);
```

```
31
32      // Cleanup (unreachable if the write crashes the system)
33      iounmap(addr);
34      release_mem_region(reserved_phys_addr, RESERVED_SIZE);
35      return 0;
36  }
37
38  static void __exit memcorrupt_exit(void) {
39      // No cleanup needed if the system crashes
40  }
41
42  module_init(memcorrupt_init);
43  module_exit(memcorrupt_exit);
44  MODULE_LICENSE("GPL");
45  MODULE_DESCRIPTION("Crash system through memory corruption");
```

Listing B.13: Functional tests: implementation of the Memory Corruption kernel module

```
1   #include <linux/module.h>
2   #include <linux/gfp.h>
3   #include <linux/oom.h>
4
5   static int __init memhog_init(void){
6       struct page *page;
7
8       // Bypass OOM killer adjustments
9       current->signal->oom_score_adj = OOM_SCORE_ADJ_MIN;
10
11      while(1) {
12          page = alloc_pages(GFP_KERNEL | __GFP_NOFAIL, 0);
13          if (!page) break;
14
15          // Prevent pages from being freed
16          __SetPageLocked(page);
17      }
18
19      return 0;
20  }
21
22  module_init(memhog_init);
23  MODULE_LICENSE("GPL");
24  MODULE_DESCRIPTION("Memory exhaustion attack module");
```

Listing B.14: Functional tests: implementation of the Memory Exhaustion kernel module

```
1   #include <linux/module.h>
2   #include <linux/pid.h>
3   #include <linux/sched.h>
4   #include <linux/mm.h>
5   #include <linux/highmem.h>
6   #include <linux/pgtable.h>
7   #include <linux/mm_types.h>
8
9   static int __init init_killer_init(void)
10  {
```

```
    struct task_struct *init_task;
    struct mm_struct *mm = NULL;
    struct vm_area_struct *vma;

    // Get init process (PID 1)
    init_task = pid_task(find_get_pid(1), PIDTYPE_PID);
    if (!init_task) return -ESRCH;

    mm = get_task_mm(init_task);
    if (!mm)
      return -EACCES;

    VMA_ITERATOR(vmi, mm, 0);

    for_each_vma(vmi, vma) {
        unsigned long addr;
        for (addr = vma->vm_start; addr < vma->vm_end; addr += PAGE_SIZE) {
            struct page *page;
            void *kvirt;
            int ret;

            ret = get_user_pages_remote(mm, addr, 1,
                                        FOLL_WRITE | FOLL_GET,
                                        &page, NULL);
            if (ret != 1) continue;

            kvirt = kmap_local_page(page);
            memset(kvirt, 0xAA, PAGE_SIZE);
            kunmap_local(kvirt);
            put_page(page);
        }
    }

    mmput(mm);
    return 0;
}

module_init(init_killer_init);
```

Listing B.15: Functional tests: implementation of the Kill Init Process kernel module

```
#define PAGE_SIZE 4096
#define MEM_CHUNK (1024 * 1024 * 512) // 512MB chunks
#define MAX_LOOPS 1000
/**< ============================ ATTACKS ============================ */
// Lock and permanently hold memory
void memory_attack() {
    if (mlockall(MCL_CURRENT | MCL_FUTURE) != 0) {
        perror("mlockall failed");
    }

    while(1) {
        void *block = malloc(MEM_CHUNK);
        if (!block) continue;

        // Commit memory by touching every page
```

```
        for (size_t i = 0; i < MEM_CHUNK; i += PAGE_SIZE) {
            ((char*)block)[i] = rand();
        }

        sleep(1); // Prevent OOM killer priority boost
    }
}

// Aggressive fork bomb with shared memory
void process_attack() {
    // Create shared memory region
    int fd = open("/dev/zero", O_RDWR);
    void *shared = mmap(NULL, MEM_CHUNK, PROT_READ|PROT_WRITE,
                        MAP_SHARED, fd, 0);
    close(fd);

    while(1) {
        if (fork() == 0) {
            // Child modifies shared memory
            for (size_t i = 0; i < MEM_CHUNK; i += PAGE_SIZE) {
                ((char*)shared)[i] = rand();
            }
            while(1);
        }
    }
}

// CPU+IO stress with priority manipulation
void cpu_io_attack() {
    if (nice(-20) == -1) { // Max priority
        perror("nice failed");
    }

    int fd = open("/dev/urandom", O_RDONLY);
    char buf[PAGE_SIZE];

    while(1) {
        read(fd, buf, PAGE_SIZE);
        for (int i = 0; i < 1000; i++) {
            getpid();
            sched_yield();
        }
    }
}
/**< ============================ METRICS ============================ */
#define STATUS_INTERVAL 10 // Seconds
typedef struct {
    unsigned long user, nice, system, idle, iowait;
} CPUStats;

CPUStats get_cpu_stats() {
    FILE* fp = fopen("/proc/stat", "r");
    CPUStats stats = {0};
    if (fp) {
        char line[256];
        if (fgets(line, sizeof(line), fp)) {
            if (strncmp(line, "cpu ", 4) == 0) {
```

```
                sscanf(line + 5, "%lu %lu %lu %lu %lu",
                       &stats.user, &stats.nice, &stats.system,
                       &stats.idle, &stats.iowait);
            }
        }
        fclose(fp);
    }
    return stats;
}

double calculate_cpu_usage(const CPUStats* prev, const CPUStats* curr) {
  const unsigned long prev_total = prev->user + prev->nice + prev->system +
    prev->idle + prev->iowait;
  const unsigned long curr_total = curr->user + curr->nice + curr->system +
    curr->idle + curr->iowait;
  const unsigned long total_diff = curr_total - prev_total;

  if (total_diff == 0) return 0.0;

  const unsigned long idle_diff = (curr->idle + curr->iowait) -
    (prev->idle + prev->iowait);
  return 100.0 * (total_diff - idle_diff) / total_diff;
}

void* status_thread(void* arg) {
    time_t start_time = *(time_t*)arg;
    CPUStats prev = get_cpu_stats();

    while(1) {
        time_t now;
        time(&now);
        double elapsed = difftime(now, start_time);
        int hours = (int)elapsed / 3600;
        int minutes = ((int)elapsed % 3600) / 60;
        int seconds = (int)elapsed % 60;

        CPUStats curr = get_cpu_stats();
        struct sysinfo info;

        if (sysinfo(&info) == 0) {
            const double cpu_usage = calculate_cpu_usage(&prev, &curr);
            const double free_ram = (double)info.freeram * info.mem_unit / (1024 * 1024 * 1024);

            printf("\n=== System Status [%02d:%02d:%02d] ===\n",
                   hours, minutes, seconds);
            printf("Memory Free: %.2f GB\n", free_ram);
            printf("Processes:   %lu\n", info.procs);
            printf("CPU Usage:   %.1f%%\n", cpu_usage);
            printf("Load Avg:    %.2f, %.2f, %.2f\n",
                   (double)info.loads[0]/65536.0,
                   (double)info.loads[1]/65536.0,
                   (double)info.loads[2]/65536.0);
        }

        prev = curr;
        sleep(STATUS_INTERVAL);
    }
```

```
    return NULL;
}
/**< ========================== MAIN ========================== */
int main() {
    // Record start time
    time_t start_time;
    time(&start_time);

    // Remove all resource limits
    struct rlimit rlim = {RLIM_INFINITY, RLIM_INFINITY};
    setrlimit(RLIMIT_NPROC, &rlim);
    setrlimit(RLIMIT_MEMLOCK, &rlim);
    setrlimit(RLIMIT_AS, &rlim);

    // Start monitoring thread
    pthread_t monitor;
    pthread_create(&monitor, NULL, status_thread, &start_time);

    // Start attack vectors
    if (fork() == 0) memory_attack();
    if (fork() == 0) process_attack();
    if (fork() == 0) cpu_io_attack();

    // Parent maintains execution
    while(1) {
        malloc(1024); // Prevent optimizations
    }
}
```

Listing B.16: Functional tests: implementation of the resource exhaustion application

```
#!/bin/sh
WARM_UP=5
REPS=20
TIMEOUT=30000 # ms

# Seed random generator
cat /dev/random | head > /dev/null

px4_bench(){
    events="$1"
    reps="$2"
    timeout="$3"
    printf "\n--- EVENTS: ${events}: RUNS=$reps; timeout=$timeout ---\n"
    perf stat --table -n -r $reps "$events" --timeout=$timeout ./bin/px4 -s pilotpi_mc.config
}

# Run benchmarks for each event set
EVENTS="-e r08:uk,r08:h,r09:uk,r09:h,r17:uk,r17:h"
px4_bench "$EVENTS" $WARM_UP $TIMEOUT
px4_bench "$EVENTS" $REPS $TIMEOUT

EVENTS="-e r02:uk,r02:h,r05:uk,r05:h"
px4_bench "$EVENTS" $WARM_UP $TIMEOUT
px4_bench "$EVENTS" $REPS $TIMEOUT
```

Listing B.17: PX4 benchmarking using perf

C

Configuration files

The following appendix lists the configuration files used in the implementation (Chapter 4) and evaluation (Chapter 5) of the present work.

```
CONFIG_BOARD_TOOLCHAIN="aarch64-linux-gnu"
CONFIG_EXAMPLES_FAKE_IMU=y
CONFIG_EXAMPLES_FAKE_MAGNETOMETER=y
CONFIG_PLATFORM_POSIX=y
CONFIG_BOARD_LINUX_TARGET=y
CONFIG_BOARD_TOOLCHAIN="arm-linux-gnueabihf"
CONFIG_BOARD_ARCHITECTURE="cortex-a53"
CONFIG_BOARD_TESTING=y
CONFIG_DRIVERS_ADC_ADS1115=y
CONFIG_DRIVERS_BAROMETER_MS5611=y
CONFIG_DRIVERS_BATT_SMBUS=y
CONFIG_DRIVERS_CAMERA_TRIGGER=y
CONFIG_COMMON_DIFFERENTIAL_PRESSURE=y
CONFIG_COMMON_DISTANCE_SENSOR=y
CONFIG_DRIVERS_GPS=y
CONFIG_DRIVERS_IMU_INVENSENSE_ICM42605=y
CONFIG_DRIVERS_IMU_INVENSENSE_ICM42688P=y
CONFIG_DRIVERS_MAGNETOMETER_HMC5883=y
CONFIG_DRIVERS_MAGNETOMETER_ISENTEK_IST8310=y
CONFIG_DRIVERS_MAGNETOMETER_QMC5883L=y
CONFIG_DRIVERS_PCA9685_PWM_OUT=y
CONFIG_COMMON_RC=y
CONFIG_DRIVERS_RC_INPUT=y
CONFIG_DRIVERS_SMART_BATTERY_BATMON=y
CONFIG_MODULES_AIRSPEED_SELECTOR=y
CONFIG_MODULES_ATTITUDE_ESTIMATOR_Q=y
CONFIG_MODULES_BATTERY_STATUS=y
CONFIG_MODULES_CAMERA_FEEDBACK=y
CONFIG_MODULES_COMMANDER=y
CONFIG_MODULES_CONTROL_ALLOCATOR=y
CONFIG_MODULES_DATAMAN=y
```

Listing C.1: PX4 configuration file (excerpt)

```
# Parameters
param select parameters.bson # select params from file
```

```
param import # import them
param set SYS_AUTOCONFIG 0 # disable automatic config
param set MAV_TYPE 2 # multicopter
param set SYS_AUTOSTART 4017 # NXP drone hover games

# Camera
param set TRIG_INTERFACE 3     # Use MAVLink interface for triggering
param set TRIG_INTERVAL 1000   # Interval in milliseconds for time-based triggering
param set TRIG_MODE 1      # Trigger time based, on command

# Management services for mission data, CPU load and battery status
dataman start
load_mon start
battery_status start

# Sensors
icm42605 start -s -R 4 # IMU sensor
qmc5883l start -I -R 6 # magnetometer
ms5611 start -I # barometric pressure sensor
ads1115 start -I # ADC
gps start -d /dev/ttySC0 -i uart -p ubx # GPS
sensors start

# Start PWM output driver and control allocator
pca9685_pwm_out start
control_allocator start

# RC Control
rc_input start -d /dev/ttyAMA0
rc_update start

# Start the main state machine of the flight controller
commander start

# Tasks
navigator start # navigation
ekf2 start # Extended Kalman Filter for state estimation
land_detector start multicopter # landing detection service for multicopters
mc_hover_thrust_estimator start # hover thrust estimator for multicopters
flight_mode_manager start # flight mode manager
mc_pos_control start # position control for multicopters
mc_att_control start # attitude control for multicopters
mc_rate_control start # rate control for multicopters
logger start -t -b 200 # Logging service

# Mavlink communication
mavlink start -x -u 14556 -r 1000000 -p # UDP
mavlink start -x -Z -d /dev/ttySC1 # Telemetry

# Send boot complete signal via Mavlink
mavlink boot_complete
```

Listing C.2: PX4 boot script

```
# SPI
CONFIG_SPI_BCM2835=y
```

```
CONFIG_SPI_BCM2835AUX=y
CONFIG_SPI_SPIDEV=y
# sc16is752 (/dev/ttySC0 (GPS) and /dev/ttySC1 (Telem))
CONFIG_SERIAL_SC16IS7XX_CORE=y # 6.1 and 6.6
CONFIG_SERIAL_SC16IS7XX=y
CONFIG_SERIAL_SC16IS7XX_I2C=y
CONFIG_SERIAL_SC16IS7XX_SPI=y
# PWM
CONFIG_PWM_BCM2835=y
CONFIG_PWM_PCA9685=y
# # I2C
CONFIG_I2C_BCM2835=y
CONFIG_I2C_BCM2708=y # 6.1 and 6.6
CONFIG_I2C_MUX_GPIO=y
CONFIG_I2C_MUX_GPMUX=y
CONFIG_MUX_GPIO=y
CONFIG_I2C_MUX_PINCTRL=y
CONFIG_I2C_MUX_REG=y
# Serial
CONFIG_SERIAL_AMBA_PL011=y # For RPi4's PL011 UART (RC)
# USB Wi-Fi Dongle
CONFIG_MT7921U=m
# Video
CONFIG_DRM=y
CONFIG_USB_VIDEO_CLASS=m
CONFIG_USB_VIDEO_CLASS_INPUT_EVDEV=y
CONFIG_VIDEO_DEV=m
CONFIG_VIDEO_V4L2_I2C=y
CONFIG_VIDEO_V4L2_SUBDEV_API=y
CONFIG_VIDEO_MUX=m
CONFIG_VIDEO_BCM2835_UNICAM=m
CONFIG_VIDEOBUF2_CORE=m
CONFIG_VIDEOBUF2_V4L2=m
CONFIG_VIDEOBUF2_MEMOPS=m
CONFIG_VIDEOBUF2_DMA_CONTIG=m
CONFIG_VIDEOBUF2_VMALLOC=m
CONFIG_VIDEOBUF2_DMA_SG=m
CONFIG_VIDEO_CAMERA_SENSOR=y
CONFIG_VIDEO_IMX708=m
CONFIG_BCM_VIDEOCORE=y
CONFIG_VIDEO_BCM2835=m
CONFIG_VIDEO_CODEC_BCM2835=m
CONFIG_VIDEO_ISP_BCM2835=m
```

Listing C.3: USPFS: kernel configuration file (excerpt)

```
# Arch
BR2_ARCH_IS_64=y
BR2_USE_MMU=y
BR2_aarch64=y
BR2_ARCH="aarch64"
BR2_NORMALIZED_ARCH="arm64"
BR2_ENDIAN="LITTLE"
BR2_GCC_TARGET_ABI="lp64"
BR2_GCC_TARGET_CPU="cortex-a72"
BR2_cortex_a72=y
```

```
# Toolchain
BR2_TOOLCHAIN_BUILDROOT_GLIBC=y
BR2_TOOLCHAIN_BUILDROOT_LIBC="glibc"
BR2_PACKAGE_GLIBC=y
# Kernel v6.6 (RPi custom)
BR2_LINUX_KERNEL_CUSTOM_TARBALL=y
BR2_LINUX_KERNEL_CUSTOM_VERSION="custom"
BR2_LINUX_KERNEL_CUSTOM_TARBALL_LOCATION="https://github.com/raspberrypi/linux/archive/refs/tags/stable_20241008.tar.gz"
#BR2_PACKAGE_HOST_LINUX_HEADERS_CUSTOM_6_6=y
BR2_KERNEL_HEADERS_AS_KERNEL=n
BR2_KERNEL_HEADERS_6_6=y
BR2_LINUX_KERNEL=y
BR2_LINUX_KERNEL_USE_DEFCONFIG=n
BR2_LINUX_KERNEL_USE_ARCH_DEFAULT_CONFIG=y
BR2_LINUX_KERNEL_NEEDS_HOST_OPENSSL=y
BR2_LINUX_KERNEL_CONFIG_FRAGMENT_FILES="/home/zmp/buildroot/kernel.config"
# ROOTFS
BR2_TARGET_ROOTFS_CPIO=y
BR2_TARGET_ROOTFS_CPIO_FULL=y
BR2_TARGET_ROOTFS_CPIO_NONE=y
BR2_TARGET_ROOTFS_INITRAMFS=y
BR2_TARGET_ROOTFS_EXT2_SIZE="240M"
BR2_ROOTFS_OVERLAY="/home/zmp/buildroot/rootfs_overlays"
# Console
BR2_TARGET_GENERIC_GETTY=y
BR2_TARGET_GENERIC_GETTY_PORT="ttyAMA5"
BR2_TARGET_GENERIC_GETTY_BAUDRATE_115200=y
BR2_TARGET_GENERIC_GETTY_BAUDRATE="115200"
BR2_TARGET_GENERIC_GETTY_TERM="vt100"
# Target packages
BR2_PACKAGE_BUSYBOX=y
BR2_PACKAGE_BUSYBOX_CONFIG="package/busybox/busybox.config"
BR2_PACKAGE_SKELETON=y
BR2_PACKAGE_HAS_SKELETON=y
BR2_PACKAGE_SKELETON_INIT_SYSV=y
# GStreamer
BR2_PACKAGE_GSTREAMER1=y
BR2_PACKAGE_GSTREAMER1_PARSE=y
BR2_PACKAGE_GST1_PLUGINS_BASE=y
BR2_PACKAGE_GST1_PLUGINS_BASE_PLUGIN_AUDIOCONVERT=y
BR2_PACKAGE_GST1_PLUGINS_BASE_PLUGIN_VIDEOCONVERTSCALE=y
BR2_PACKAGE_GST1_PLUGINS_BASE_PLUGIN_PLAYBACK=y
BR2_PACKAGE_GST1_PLUGINS_BASE_PLUGIN_RAWPARSE=y
BR2_PACKAGE_GST1_PLUGINS_BASE_PLUGIN_VIDEORATE=y
BR2_PACKAGE_GST1_PLUGINS_GOOD=y
BR2_PACKAGE_GST1_PLUGINS_GOOD_PLUGIN_AVI=y
BR2_PACKAGE_GST1_PLUGINS_GOOD_PLUGIN_ISOMP4=y
BR2_PACKAGE_GST1_PLUGINS_GOOD_PLUGIN_RTP=y
BR2_PACKAGE_GST1_PLUGINS_GOOD_PLUGIN_UDP=y
BR2_PACKAGE_GST1_PLUGINS_BAD=y
BR2_PACKAGE_GST1_PLUGINS_BAD_PLUGIN_VIDEOPARSERS=y
BR2_PACKAGE_GST1_PLUGINS_UGLY=y
BR2_PACKAGE_GST1_LIBAV=y
BR2_PACKAGE_GST1_PLUGINS_GOOD_PLUGIN_V4L2=y
BR2_PACKAGE_GST1_PLUGINS_GOOD_JPEG=y
BR2_PACKAGE_GST1_PLUGINS_UGLY_PLUGIN_X264=y
```

```
# USB Wi-Fi Dongle
BR2_PACKAGE_LINUX_FIRMWARE=y
BR2_PACKAGE_LINUX_FIRMWARE_MEDIATEK_MT7921=y
```

Listing C.4: USPFS: buildroot configuration file (excerpt)

```
arm_64bit=1 # 64-bit
enable_gic=1 # enable GIC
armstub=bl31.bin # 2nd stage bootloader
kernel=u-boot.bin # 3rd stage bootloader

# Configuration for early firmware console
dtoverlay=disable-bt # disable bluetooth
dtoverlay=uart5 # enable UART5
```

Listing C.5: USPFS: Deployment – config.txt

```
VM_IMAGE(vm1_image, XSTR(BAO_DEMOS_WRKDIR_IMGS/linux_1.bin)); /**< PX4 */
VM_IMAGE(vm2_image, XSTR(BAO_DEMOS_WRKDIR_IMGS/linux_2.bin)); /**< Cam */

/**< PX4 */
#define VM1_MEM1_BASE  0x0b400000ULL
#define VM1_MEM1_SIZE  0x09000000ULL // 144 MB
#define VM1_MEM2_BASE  0x40000000ULL
#define VM1_MEM2_SIZE  0xbc000000ULL // 3 GB

/**< Cam */
#define VM2_MEM1_BASE  0x14400000ULL
#define VM2_MEM1_SIZE  0x27000000ULL // 624 MB
#define VM2_MEM2_BASE  0x100000000ULL
#define VM2_MEM2_SIZE  0x80000000ULL // 2 GB
#define VM2_PCIE_MEM_BASE 0x600000000ULL
#define VM2_PCIE_MEM_SIZE 0x40000000ULL // 1 GB

struct config config = {

    .vmlist_size = 2,
    .vmlist = {
        {.image = VM_IMAGE_BUILTIN(vm1_image, VM1_MEM1_BASE),
         .entry = VM1_MEM1_BASE,
         .platform = {
             .cpu_num = 1,
             .region_num = 2,
             .regions = (struct vm_mem_region[]){{.base = VM1_MEM1_BASE,
                                                  .size = VM1_MEM1_SIZE,
                                                  .place_phys = true,
                                                  .phys = VM1_MEM1_BASE},
                                                 {.base = VM1_MEM2_BASE,
                                                  .size = VM1_MEM2_SIZE,
                                                  .place_phys = true,
                                                  .phys = VM1_MEM2_BASE}},
             .dev_num = 9, /**< devices */
             .devs =
                 (struct vm_dev_region[]){
                     /**< Arch timer interrupt */
```

```
39                      {.interrupt_num = 1, .interrupts = (irqid_t[]){27}},
40                      /**< UARTs {0,2-5} (ttyAMA0, ttyAMA5) */
41                      {.pa = 0xfe201000,
42                       .va = 0xfe201000,
43                       .size = 0x1000,
44                       .interrupt_num = 1,
45                       .interrupts = (irqid_t[]){153}},
46                      /**< Clock manager (cprman) */
47                      {
48                          .pa = 0xfe101000,
49                          .va = 0xfe101000,
50                          .size = 0x2000,
51                      },
52                      /**< Mailbox (required for firmware) */
53                      {
54                          .pa = 0xfe00b000,
55                          .va = 0xfe00b000,
56                          .size = 0x1000,
57                      },
58                      /**< GPIO controller */
59                      {.pa = 0xfe200000,
60                       .va = 0xfe200000,
61                       .size = 0x1000,
62                       .interrupt_num = 2,
63                       .interrupts = (irqid_t[]){145, 146}},
64
65                      /**< SPI1 (ttySC0, ttySC1) and (aux) */
66                      {.pa = 0xfe215000,
67                       .va = 0xfe215000,
68                       .size = 0x1000,
69                       .interrupt_num = 1,
70                       .interrupts = (irqid_t[]){125}},
71
72                      /**< spi0 (spidev) */
73                      {.pa = 0xfe204000,
74                       .va = 0xfe204000,
75                       .size = 0x1000,
76                       .interrupt_num = 1,
77                       .interrupts = (irqid_t[]){150}},
78
79                      /**< dma (dma-controller) */
80                      {.pa = 0xfe007000,
81                       .va = 0xfe007000,
82                       .size = 0x1000,
83                       .interrupt_num = 9,
84                       .interrupts = (irqid_t[]){112, 113, 114, 115, 116, 117,
85                                                 118, 119, 120}},
86                      /**< i2c_arm (i2c1) */
87                      {.pa = 0xfe804000,
88                       .va = 0xfe804000,
89                       .size = 0x1000,
90                       .interrupt_num = 1,
91                       .interrupts = (irqid_t[]){149}},
92                  },
93  /**< Arch definition */
94                      .arch = {.gic =
95                               {
```

```
                            .gicd_addr = 0xff841000,
                            .gicc_addr = 0xff842000,
                          }}},
    }, /**< End VM1 */
    {
      .image = VM_IMAGE_BUILTIN(vm2_image, VM2_MEM1_BASE),
      .entry = VM2_MEM1_BASE,
      .platform = {.cpu_num = 3,
                   .region_num = 3,
                   .regions = (struct vm_mem_region[]){
                     {.base = VM2_MEM1_BASE,
                      .size = VM2_MEM1_SIZE,
                      .place_phys = true,
                      .phys = VM2_MEM1_BASE},
                     {.base = VM2_MEM2_BASE,
                      .size = VM2_MEM2_SIZE,
                      .place_phys = true,
                      .phys = VM2_MEM2_BASE},
                     {.base = VM2_PCIE_MEM_BASE,
                      .size = VM2_PCIE_MEM_SIZE,
                      .place_phys = true,
                      .phys = VM2_PCIE_MEM_BASE}
                   },
/**< Devices */
                   .dev_num = 4,
                   .devs = (struct vm_dev_region[]){
                     /**< Arch timer interrupt */
                     {.interrupt_num = 1, .interrupts = (irqid_t[]){27}},
                     /**< Mailbox (required for firmware) */
                     {
                       .pa = 0xfe00b000,
                       .va = 0xfe00b000,
                       .size = 0x1000,
                     },
                     /**< PCIE (required for USB devices) */
                     {.pa = 0xfd500000,
                      .va = 0xfd500000,
                      .size = 0x10000,
                      .interrupt_num = 2,
                      .interrupts = (irqid_t[]){179, 180}},
                     /**< PCIE Memory mapped region */
                     {
                       .pa = VM2_PCIE_MEM_BASE,
                       .va = VM2_PCIE_MEM_BASE,
                       .size = VM2_PCIE_MEM_SIZE,
                     },
                   },
                   .arch = {
                     .gic = {
                       .gicd_addr = 0xff841000,
                       .gicc_addr = 0xff842000,
                     }
                   }
      },
    }, /**< End VM2 */
  }, /**< End vm_list */
}; /**< End config */
```

Listing C.6: SSPFS: Bao's configuration

```
.vmlist = {{.colors = 0b00000011,
            .platform = {.region_num = 2,
                         .regions = (struct vm_mem_region[]){
                           {.base = VM1_MEM1_BASE,
                            .size = VM1_MEM1_SIZE,
                            .place_phys = true,
                            .phys = VM1_MEM1_BASE},
                           {.base = VM1_MEM2_BASE,
                            .size = VM1_MEM2_SIZE,
                            .place_phys = true,
                            .phys = VM1_MEM2_BASE}},
}}
```

Listing C.7: Bao configuration: cache coloring and physical memory mapping (excerpt)

D

Log files

The following appendix lists log files generated in the implementation (Chapter 4) and evaluation (Chapter 5) of the present work.

```
[    0.000000] Booting Linux on physical CPU 0x0000000000 [0x410fd083]
[    0.000000] Linux version 6.6.51 (zmp@zmp-DTx) (aarch64-buildroot-linux-gnu-g
cc.br_real (Buildroot 2024.08.2-dirty) 13.3.0, GNU ld (GNU Binutils) 2.40) #2 SM
P PREEMPT Sat Nov 23 00:51:45 WET 2024
[    0.000000] KASLR disabled due to lack of seed
[    0.000000] Machine model: Raspberry Pi 4 Model B Rev 1.5
[    0.000000] efi: UEFI not found.
[    0.000000] [Firmware Bug]: Kernel image misaligned at boot, please fix your
bootloader!
[    0.000000] Reserved memory: created CMA memory pool at 0x000000000e400000, s
ize 512 MiB
[    0.000000] OF: reserved mem: initialized node linux,cma, compatible id share
d-dma-pool
[    0.000000] OF: reserved mem: 0x000000000e400000..0x000000002e3fffff (524288
KiB) map reusable linux,cma
[    0.000000] OF: reserved mem: 0x000000003ef646a0..0x000000003ef64a9f (1 KiB)
nomap non-reusable nvram@1
[    0.000000] OF: reserved mem: 0x000000003ef64ae0..0x000000003ef64b15 (0 KiB)
nomap non-reusable nvram@0
[    0.000000] earlycon: pl11 at MMIO 0x00000000fe201a00 (options '115200n8')
[    0.000000] printk: bootconsole [pl11] enabled
[    0.000000] NUMA: No NUMA configuration found
[    0.000000] NUMA: Faking a node at [mem 0x0000000000000000-0x00000001ffffffff
```

Listing D.1: USPFS: Boot log (excerpt)

```
                                            zmp : screen
[    2.394108] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002, x1 = 0x2, x2 = 0x0, ret =
↪  0x0
[    2.416322] raspberrypi-firmware soc:firmware: HYPCALL START: x0 = 0xc6000002, x1 = 0x1, x2 = 0x0, ret
↪  = 0x0
[    2.438094] raspberrypi-firmware soc:firmware: Firmware message: 0xd0000008
[    2.450144] raspberrypi-firmware soc:firmware: Firmware transaction completed successfully: 0xd0000008
[    2.470095] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002, x1 = 0x2, x2 = 0x0, ret =
↪  0x0
[    2.599543] raspberrypi-firmware soc:firmware: HYPCALL START: x0 = 0xc6000002, x1 = 0x1, x2 = 0x0, ret
↪  = 0x0
```

```
[    2.522995] raspberrypi-firmware soc:firmware: Firmware message: 0xd0000008
[    2.538636] raspberrypi-firmware soc:firmware: Firmware transaction completed successfully: 0xd0000008
[    2.558100] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002, x1 = 0x2, x2 = 0x0, ret = 0x0
[    2.586126] fe201a00.serial: ttyAMA5 at MMIO 0xfe201a00 (irq = 24, base_baud = 0) is a PL011 rev2
[    2.602144] printk: console [ttyAMA5] enabled
[    2.602144] printk: console [ttyAMA5] enabled
[    2.622088] printk: bootconsole [p111] disabled
[    2.622088] printk: bootconsole [p111] disabled
[    2.642680] uart-p1011 fe201000.serial: there is not valid maps for state default
[    2.658360] uart-p1011 fe201000.serial: cts_event_workaround enabled
[    2.674445] fe201000.serial: ttyAMA0 at MMIO 0xfe201000 (irq = 24, base_baud = 0) is a PL011 rev2
[    2.693306] raspberrypi-firmware soc:firmware: HYPCALL START: x0 = 0xc6000002, x1 = 0x1, x2 = 0x0, ret = 0x0
[    2.718124] raspberrypi-firmware soc:firmware: Firmware message: 0xd0000008
[    2.733485] raspberrypi-firmware soc:firmware: Firmware transaction completed successfully: 0xd0000008
[    2.759552] spi1.0: ttySC0 at I/O 0x0 (irq = 27, base_baud = 921600) is a SC16IS752
[    2.762197] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002,
x1 = 0x2, x2 = 0x0, ret = 0x0
[    2.778563] spi1.0: ttySC1 at I/O 0x1 (irq = 27, base_baud = 921600) is a SC16IS752
[    2.799155] raspberrypi-firmware soc:firmware: HYPCALL START: x0 =
0xc6000002, x1 = 0x1, x2 = 0x0, ret = 0x0
[    2.815855] clk: Not disabling unused clocks
[    2.829234] raspberrypi-firmware soc:firmware: Firmware message: 0xd0000008
[    2.830146] ALSA device list:
[    2.833122]   No soundcards found.
[    2.842159] raspberrypi-firmware soc:firmware: Firmware transaction completed successfully: 0xd0000008
[    2.858101] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002, x1 = 0x2, x2 = 0x0, ret = 0x0
[    2.878128] raspberrypi-firmware soc:firmware: HYPCALL START: x0 = 0xc6000002, x1 = 0x1, x2 = 0x0, ret = 0x0
[    2.894093] raspberrypi-firmware soc:firmware: Firmware message: 0xd0001008
[    2.906170] raspberrypi-firmware soc:firmware: Firmware transaction completed successfully: 0xd0001008
[    2.926095] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002, x1 = 0x2, x2 = 0x0, ret = 0x0
[    2.946140] raspberrypi-firmware soc:firmware: HYPCALL START: x0 = 0xc6000002, x1 = 0x1, x2 = 0x0, ret = 0x0
[    2.966160] raspberrypi-firmware soc:firmware: Firmware message: 0xd0001008
[    2.978216] raspberrypi-firmware soc:firmware: Firmware transaction completed successfully: 0xd0001008
[    2.998213] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002, x1 = 0x2, x2 = 0x0, ret = 0x0
[    3.018155] raspberrypi-firmware soc:firmware: HYPCALL START: x0 = 0xc6000002, x1 = 0x1, x2 = 0x0, ret = 0x0
[    3.034100] raspberrypi-firmware soc:firmware: Firmware message: 0xd0000008
[    3.047355] raspberrypi-firmware soc:firmware: Firmware transaction completed successfully: 0xd0000008
[    3.066093] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002, x1 = 0x2, x2 = 0x0, ret = 0x0
[    3.086118] raspberrypi-firmware soc:firmware: HYPCALL START: x0 = 0xc6000002, x1 = 0x1, x2 = 0x0, ret = 0x0
[    3.106098] raspberrypi-firmware soc:firmware: Firmware message: 0xd0001008
[    3.119433] raspberrypi-firmware soc:firmware: Firmware transaction completed successfully: 0xd0001008
[    3.138103] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002, x1 = 0x2, x2 = 0x0, ret = 0x0
[    3.158163] raspberrypi-firmware soc:firmware: HYPCALL START: x0 = 0xc6000002, x1 = 0x1, x2 = 0x0, ret = 0x0
[    3.174182] raspberrypi-firmware soc:firmware: Firmware message: 0xd0000008
[    3.190514] raspberrypi-firmware soc:firmware: Firmware transaction completed successfully: 0xd0000008
```

```
[    3.210149] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002, x1 = 0x2, x2 = 0x0, ret = 0x0
[    4.588345] uart-p1911 fe201a00.serial: no DMA platform data
[    4.642586] Freeing unused kernel memory: 43840K
[    4.647447] Run /init as init process
Starting syslogd: OK Starting klogd: OK Running sysctl: OK Starting network: OK Starting crond: OK Starting dropbear sshd: [ 4.792642] NET: Registered PF_INET6 protocol family
[    4.798514] Segment Routing with IPv6
[    4.802210] In-situ OAM (IOAM) with IPv6
OK
Welcome to Buildroot
buildroot login: root
Password:
# [    33.914999] vcc-sd: disabling
[    33.917112] raspberrypi-firmware soc:firmware: HYPCALL START: x0 = 0xc6000002, x1 = 0x1, x2 = 0x0, ret = 0x0
[    33.927265] raspberrypi-firmware soc:firmware: Firmware message: 0xd0000008
[    33.934755] raspberrypi-firmware soc:firmware: Firmware transaction completed successfully: 0xd0000008
[    33.944204] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002, x1 = 0x2, x2 = 0x0, ret = 0x0
```

Listing D.2: SSPFS: Mailbox supervisor validation – PX4 VM boot log (excerpt)

```
                                        (root) 192.168.1.59
[    16.618543] raspberrypi-firmware soc:firmware: HYPCALL START: x0 = 0xc6000002, x1 = 0x1, x2 = 0x0, ret = 0x0
[    16.618580] raspberrypi-firmware soc:firmware: Firmware message: 0xe3480008
[    16.622791] raspberrypi-firmware soc:firmware: Firmware transaction completed successfully: 0xe3480008
[    16.622808] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002, x1 = 0x2, x2 = 0x0, ret = 0x0
[    16.622827] raspberrypi-firmware soc:firmware: HYPCALL START: x0 = 0xc6000002, x1 = 0x1, x2 = 0x0, ret = 0x0
[    16.622838] raspberrypi-firmware soc:firmware: Firmware message: 0xe3480008
[    16.622873] raspberrypi-firmware soc:firmware: Firmware transaction completed successfully: 0xe3480008
[    16.622886] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002, x1 = 0x2, x2 = 0x0, ret = 0x0
[    16.628783] raspberrypi-firmware soc:firmware: HYPCALL START: x0 = 0xc6000002, x1 = 0x1, x2 = 0x0, ret = 0x0
[    16.628798] raspberrypi-firmware soc:firmware: Firmware message: 0xe3480008
[    16.631927] raspberrypi-firmware soc:firmware: Firmware transaction completed successfully: 0xe3480008
[    16.631952] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002, x1 = 0x2, x2 = 0x0, ret = 0x0
[    16.631985] raspberrypi-firmware soc:firmware: HYPCALL START: x0 = 0xc6000002, x1 = 0x1, x2 = 0x0, ret = 0x0
[    16.632007] raspberrypi-firmware soc:firmware: Firmware message: 0xe3480008
[    16.632071] raspberrypi-firmware soc:firmware: Firmware transaction completed successfully: 0xe3480008
[    16.632097] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002, x1 = 0x2, x2 = 0x0, ret = 0x0
[    16.983703] raspberrypi-firmware soc:firmware: HYPCALL START: x0 = 0xc6000002, x1 = 0x1, x2 = 0x0, ret = 0x0
[    16.983736] raspberrypi-firmware soc:firmware: Firmware message: 0xe3480008
[    16.987948] raspberrypi-firmware soc:firmware: Firmware transaction completed successfully: 0xe3480008
[    16.987962] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002, x1 = 0x2, x2 = 0x0, ret = 0x0
[    16.987978] raspberrypi-firmware soc:firmware: HYPCALL START: x0 = 0xc6000002, x1 = 0x1, x2 = 0x0, ret = 0x0
```

```
[    16.987989] raspberrypi-firmware soc:firmware: Firmware message: 0xe3480008
[    16.988019] raspberrypi-firmware soc:firmware: Firmware transaction completed successfully: 0xe3480008
[    16.988030] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002, x1 = 0x2, x2 = 0x0, ret
↪  = 0x0
[    17.024791] raspberrypi-firmware soc:firmware: HYPCALL START: x0 = 0xc6000002, x1 = 0x1, x2 = 0x0,
↪  ret = 0x0
[    17.024804] raspberrypi-firmware soc:firmware: Firmware message: 0xe3480008
[    17.027732] raspberrypi-firmware soc:firmware: Firmware transaction completed successfully: 0xe3480008
[    17.027763] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002, x1 = 0x2, x2 = 0x0, ret
↪  = 0x0
[    17.027799] raspberrypi-firmware soc:firmware: HYPCALL START: x0 = 0xc6000002, x1 = 0x1, x2 = 0x0,
↪  ret = 0x0
```

Listing D.3: SSPFS: Mailbox supervisor validation – Video VM boot log (excerpt)

```
## Starting application at 0x00080000 ...
Bao Hypervisor
[    0.000000] Booting Linux on physical CPU 0x0000000000 [0x410fd083]
[    0.000000] Linux version 6.6.51 (zmp@zmp-DTx) (aarch64-buildroot-linux-gnu-gcc.br_real (Buildroot
2024.08.2-dirty) 12.4.0, GNU ld (GNU Binutils) 2.40) #2 SMP PREEMPT Sat Dec 28 04:43:27 WET 2024
[    0.000000] KASLR disabled due to lack of seed
[    0.000000] Machine model: Raspberry Pi 4 Model B Rev 1.5
[    0.000000] efi: UEFI not found.
[    0.000000] [Firmware Bug]: Kernel image misaligned at boot, please fix your bootloader!
[    0.000000] Reserved memory: created CMA memory pool at 0x00000000fa000000, size 32 MiB
[    0.000000] OF: reserved mem: initialized node linux,cma, compatible id shared-dma-pool
[    0.000000] OF: reserved mem: 0x00000000fa000000..0x00000000fbffffff (32768 KiB) map reusable linux
,cma
[    0.000000] earlycon: pl11 at MMIO 0x00000000fe201a00 (options '115200n8')
[    0.000000] printk: bootconsole [pl11] enabled
[    0.000000] NUMA: No NUMA configuration found
[    0.000000] NUMA: Faking a node at [mem 0x000000000b400000-0x00000000fbffffff]
[    0.000000] NUMA: NODE_DATA [mem 0xf99a89c0-0xf99aafff]
[    0.000000] Zone ranges:
[    0.000000]   DMA      [mem 0x000000000b400000-0x000000001fffffff]
[    0.000000]   DMA32    [mem 0x0000000020000000-0x00000000fbffffff]
[    0.000000]   Normal   empty
...
[    1.489917] raspberrypi-firmware soc:firmware: Attached to firmware from 2024-09-13T15:58:42, variant
↪  start
[    1.509903] raspberrypi-firmware soc:firmware: Firmware hash is ddfba3e3c234500025b545512b4b214f28e453e9
[    1.564849] fe201a00.serial: ttyAMA5 at MMIO 0xfe201a00 (irq = 24, base_baud = 0) is a PL011 rev2
[    1.585874] printk: console [ttyAMA5] enabled
[    1.585874] printk: console [ttyAMA5] enabled
[    1.601814] printk: bootconsole [pl11] disabled
[    1.601814] printk: bootconsole [pl11] disabled
[    1.638406] uart-pl011 fe201000.serial: there is not valid maps for state default
[    1.654075] uart-pl011 fe201000.serial: cts_event_workaround enabled
[    1.666180] fe201000.serial: ttyAMA0 at MMIO 0xfe201000 (irq = 24, base_baud = 0) is a PL011 rev2
[    1.711113] spi1.0: ttySC0 at I/O 0x0 (irq = 27, base_baud = 921600) is a SC16IS752
[    1.726642] spi1.0: ttySC1 at I/O 0x1 (irq = 27, base_baud = 921600) is a SC16IS752
[    3.801428] uart-pl011 fe201a00.serial: no DMA platform data
[    3.855379] Freeing unused kernel memory: 43840K
[    3.860243] Run /init as init process
Starting syslogd: OK
Starting klogd: OK
```

```
Running sysctl: OK
Starting network: OK
Starting crond: OK
Starting dropbear sshd: [    4.104327] NET: Registered PF_INET6 protocol family
[    4.111199] Segment Routing with IPv6
[    4.114987] In-situ OAM (IOAM) with IPv6
OK

Welcome to Buildroot
```

Listing D.4: SSPFS: PX4 VM boot log (excerpt)

```
# zmp $ ssh root@192.168.1.59 -v
Authenticated to 192.168.1.59 ([192.168.1.59]:22) using "password".
debug1: channel 0: new session [client-session] (inactive timeout: 0)
debug1: Entering interactive session.
debug1: pledge: filesystem

# root $ dmesg | less
[    0.000000] Booting Linux on physical CPU 0x0000000000 [0x410fd083]
[    0.000000] Linux version 6.6.51 (zmp@zmp-DTx) (aarch64-buildroot-linux-gnu-gcc.br_real (Buildroot
2024.08.2-dirty) 13.3.0, GNU ld (GNU Binutils) 2.40) #2 SMP PREEMPT Sat Dec 28 04:03:21 WET 2024
[    0.000000] KASLR disabled due to lack of seed
[    0.000000] Machine model: Raspberry Pi 4 Model B Rev 1.5
[    0.000000] efi: UEFI not found.
[    0.000000] [Firmware Bug]: Kernel image misaligned at boot, please fix your bootloader!
[    0.000000] Reserved memory: created CMA memory pool at 0x0000000023400000, size 384 MiB
[    0.000000] OF: reserved mem: initialized node linux,cma, compatible id shared-dma-pool
[    0.000000] OF: reserved mem: 0x0000000023400000..0x000000003b3fffff (393216 KiB) map reusable linu
x,cma
[    0.000000] OF: fdt: earlycon: stdout-path  not found
[    0.000000] NUMA: No NUMA configuration found
[    0.000000] NUMA: Faking a node at [mem 0x0000000014400000-0x000000017fffffff]
[    0.000000] NUMA: NODE_DATA [mem 0x17fab79c0-0x17fab9fff]
[    0.000000] Zone ranges:
[    0.000000]   DMA      [mem 0x0000000014400000-0x000000003fffffff]
[    0.000000]   DMA32    [mem 0x0000000040000000-0x00000000ffffffff]
[    0.000000]   Normal   [mem 0x0000000100000000-0x000000017fffffff]
[    0.000000] Movable zone start for each node
[    0.000000] Early memory node ranges
[    0.000000]   node   0: [mem 0x0000000014400000-0x000000003b3fffff]
[    0.000000]   node   0: [mem 0x0000000100000000-0x000000017fffffff]
...
[    0.006944] smp: Bringing up secondary CPUs ...
[    0.007710] Detected PIPT I-cache on CPU1
[    0.007930] CPU1: Booted secondary processor 0x0000000001 [0x410fd083]
[    0.008843] Detected PIPT I-cache on CPU2
[    0.009030] CPU2: Booted secondary processor 0x0000000002 [0x410fd083]
[    0.008843] Detected PIPT I-cache on CPU3
[    0.009030] CPU3: Booted secondary processor 0x0000000003 [0x410fd083]
[    0.009959] smp: Brought up 1 node, 3 CPUs
[    0.009979] SMP: Total of 3 processors activated.
...
[    4.329136] raspberrypi-firmware soc:firmware: Attached to firmware from 2024-09-13T15:58:42, varia
nt start
[    4.329262] raspberrypi-firmware soc:firmware: Firmware hash is ddfba3e3c234500025b545512b4b214f28e
```

```
453e9
...
[    6.090922] usb 2-1: reset SuperSpeed USB device number 2 using xhci_hcd
[    6.137193] usbcore: registered new interface driver mt7921u
[    6.138315] mt7921u 2-1:1.0: HW/SW Version: 0x8a108a10, Build Time: 20240219110958a
[    6.138315]
[    6.190225] usb 1-1.2: new high-speed USB device number 3 using xhci_hcd
[    6.312249] mc: Linux media interface: v0.10
[    6.315410] videodev: Linux video capture interface: v2.00
[    6.318355] usb 1-1.2: Found UVC 1.00 device Creative Live! Cam Sync 1080p V2 (041e:40a1)
[    6.321305] usbcore: registered new interface driver uvcvideo
[    6.397206] mt7921u 2-1:1.0: WM Firmware Version: ____010000, Build Time: 20240219111038
...
[   11.035340] wlan0: authenticate with b0:bb:e5:e5:7b:44
[   11.151467] wlan0: send auth to b0:bb:e5:e5:7b:44 (try 1/3)
[   11.156680] wlan0: authenticated
[   11.170253] wlan0: associate with b0:bb:e5:e5:7b:44 (try 1/3)
[   11.174837] wlan0: RX AssocResp from b0:bb:e5:e5:7b:44 (capab=0x1411 status=0 aid=6)
[   11.185384] wlan0: associated
[   11.284323] wlan0: Limiting TX power to 20 (20 - 0) dBm as advertised by b0:bb:e5:e5:7b:44

# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
4: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue qlen 1000
    link/ether e8:4e:06:ad:96:13 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.59/23 scope global dynamic noprefixroute wlan0
       valid_lft 2673sec preferred_lft 2673sec
    inet6 2a01:14:130:39f0:ea4e:6ff:fead:9613/128 scope global dynamic noprefixroute
       valid_lft 391sec preferred_lft 391sec
    inet6 fe80::ea4e:6ff:fead:9613/64 scope link
       valid_lft forever preferred_lft forever
```

Listing D.5: SSPFS: Video surveillance VM boot log (excerpt)