1 2 9 0

UNIVERSIDADE D
COIMBRA

Artur Duarte Coutinho

Fault Injector for Autonomous Quadrotors

Dissertation in the context of the Master in Informatics Engineering, Specialization in Software Engineering advised by Prof. Henrique Madeira & Dr. Naghmeh Ivaki and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering

June 2021

This page is intentionally left blank.

Faculty of Sciences and Technology

Department of Informatics Engineering

Fault Injector for Autonomous Quadrotors

Artur Duarte Coutinho

Dissertation in the context of the Master in Informatics Engineering, Specialization in Software Engineering advised Prof. Henrique Madeira & Dr. Naghmeh Ivaki and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering.

June 2021

1 2 9 0

UNIVERSIDADE Ð COIMBRA

This page is intentionally left blank.

Abstract

Presently, there is a constant increase in interest in UAVs due to their versatile capabilities and the wide array of tasks that have been reinvented with their usage, from delivery services to military operations. Ensuring the safety of the missions in which they are involved is especially important, considering the scalability of these missions and, consequently, the increase in possible failure scenarios. Although safety is a primary concern, commercial UAVs do not have complex fault tolerance capabilities that are available in manned aircraft. As a result, the possible benefits that UAVs may bring in the future, are hindered by the lack of uniform safety regulations and the means to define them.

This work intends to address the stated issue, by providing insight on the design and implementation of a fault injection tool to be used for the assessment of the fault-tolerant mechanisms and systems in place in autonomous quadrotors, covering all the steps necessary to provide such assessment. The tool is expected to aid said assessment, by providing a simulated, yet realistic, environment that may be used to gauge commercial UAVs. It is our goal to create a tool that supports a multitude of distinct UAVs and their respective software, in order to centralize and uniformize safety assessment techniques.

We also aim to show the effectiveness of the fault injector in the evaluation of fault tolerance processes and fault-tolerant mechanisms in drones. This is accomplished by making use of self-implemented faults, within the defined environment, and their respective analysis.

Keywords

Autonomous Quadcopter, Safety, Fault Tolerance, Fault Model, Fault Injection

This page is intentionally left blank.

Resumo

Actualmente, dadas as capacidades que os *UAVs* possuem e a grande variedade de missões que foram reinventadas com o uso dos mesmos, desde serviços de entrega a operações militares, o interesse neles tem aumentado. Assegurar a segurança de missões que envolvem *UAVs* é especialmente importante, dada a escalabilidade destas missões e, consequentemente, o aumento de possíveis cenários de falha. Apesar disto, os *UAVs* comerciais não dispõem, em geral, das capacidades necessárias para tolerar possíveis falhas, ao contrário de aviações tripuladas. Isto implica que os possíveis benefícios que os *UAVs* podem trazer no futuro são prejudicados pela falta de regulamentos de segurança uniformes e dos meios para defini-los.

Este trabalho pretende resolver o problema previamente apresentado, desenvolvendo uma ferramenta de injecção de falhas num sistema de quadricópteros autónomos e demonstrando todos os passos necessários para a sua utilização na avaliação do impacto das falhas em drones. É esperado que a ferramenta auxilie esta avaliação, providenciando um ambiente simulado, mas realista, que possa ser usado para medir a eficácia de *UAVs* comerciais. O nosso objectivo é criar uma ferramenta que suporte uma vasta quantidade de modelos de *UAVs* diferentes, e os seus respectivos software, de modo a que se possa centralizar e uniformizar um conjunto de tácticas de avaliação de segurança.

Pretendemos também demonstrar a eficácia do injector de falhas desenvolvido na avaliação de processos e mecanismos de tolerância a falhas em drones. Isto é realizado usando uma variedade de falhas, implementadas por nós dentro do ambiente definido, e a análise das mesmas.

Palavras-Chave

Quadricóptero Autónomo, Segurança, Tolerância de Falhas, Modelo de Falhas, Injeção de Falhas

This page is intentionally left blank.

Contents

This page is intentionally left blank.

Acronyms

DoS Denial of service. 11

ROS Robot Operating System. 20, 21, 23, 24

SUA System Under Assessment. 1, 14, 22, 46, 48

UAS Unmanned Aircraft System. 2, 4, 6, 7, 9, 10, 17, 52

UAV Unmanned Aerial Vehicle. 1, 2, 4, 5, 8–12, 17, 21, 24

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

Presently, there is a constant increase in interest in UAVs [3], due to their versatile capabilities and the wide array of tasks and applications that have been reinvented with their usage, from delivery services [4] to military operations [5]. Although they have brought modern solutions to various areas, they have also brought serious concerns regarding **safety** (and also security and privacy) issues. Laws, regulations, and standards that deal with these issues in UAVs have yet to be properly defined and implemented due to the recency of these vehicles, their quick growth in the last couple of years and their versatility.

Although fixed-wing UAVs are also sought, mostly for aerial surveys, rotatory-wing UAVs, particularly quadrotors, are the ones that have been increasing in representation throughout the world. This is due to the flexibility in movement that a rotatory-wing system presents. Even though this type of UAV is sought for many applications, a great amount of those applications will require a group of collaborative autonomous UAVs in order to accomplish their missions. Assuring the safety of these vehicles is especially important considering the fact that an increase in the applications and quantity of vehicles involved, will also lead to an increase in the number of possible failure scenarios, with consequent increase of safety issues.

In general, two main methods can be used to increase the dependability of these systems towards faults: *fault avoidance* and *fault tolerance*. Unfortunately, due to the growing complexity of computer-driven systems in the last years, fault avoidance processes are usually not enough [6], and as such, in order to maintain systems dependable (i.e., more focused on safety in this work), it is necessary to use fault tolerance processes/techniques.

To further verify and validate the effectiveness of the existing fault tolerance processes/techniques and the newly proposed ones in autonomous quadcopters, this work aims to provide insight into the definition and development of a fault injection technique/tool and all the necessary steps to achieve it. These steps include: i) definition and characterization of the system under assessment (SUA) and its environment (i.e., context and scope of the work); ii) identification and characterization of failure scenarios; iii) creation of a representative and possibly complete fault model (i.e., including both generic and SUA's specific faults); iv) design and implementation of fault injection technique and tool that can be served in diverse types of quadcopters.

The rest of this document is organized as follows. Chapter 2 provides the reader with the background information necessary to understand the remaining chapters. It provides an explanation on the dependability and safety related concepts that were deemed essential and insight on other works that share similar goals or topics as ours. In Chapter 3, we present the principal objectives for our work and the followed approach in order to achieve them. Chapter 4 provides some general context, based on real applications, which the work uses as a basis. These applications are based on project BUBBLES, a project targeting the formulation and validation of a concept of separation management for UAS in the U-space. Chapter 5 is dedicated to explaining the decisions made in regard to the chosen representative environment – the reasoning behind the choice of software and a detailed view on the simulated UAV components. It also covers the failure scenarios identified in the context of our environment, as well as present the fault model developed from said scenarios. Chapter 6 offers an in-depth explanation of the tool's conceptualization and implementation, while making use of a great amount of visual support. Chapter 7 introduces the reader to the experiment and metrics used to validate the faults implemented in conjunction with the fault injector. In Chapter 8, we analyze the results obtained from the experiment presented and validate the implemented faults. Lastly, Chapter 9 presents our conclusions and expectations for the future.

This page is intentionally left blank.

Chapter 2

Background and Related Work

2.1 Introduction to Drones

The Oxford Dictionary of English describes a drone as a remote-controlled pilotless aircraft [7], or in other words, as an **unmanned aerial vehicle (UAV)**. As UAVs cannot fully operate by themselves, they are only a component within a more extensive system, the **unmanned aircraft system (UAS)**. Although the UAV can be differentiated from the UAS, a great amount of works use these terms interchangeably. The UASs are made up of the following four principal components [8]:

- A UAV, composed by both hardware - namely a body, a power supply, a flight controller board and the necessary actuators - and software - mainly the flight stack which can be composed by a firmware, a middleware and an operating system;
- A control station from which the pilot interacts with the UAV to control it;
- A communication equipment required to establish communication between the UAV and the control station;
- Payloads (e.g., sensors, GPS, etc.) added to the UAV that may vary depending on the tasks to be accomplished.

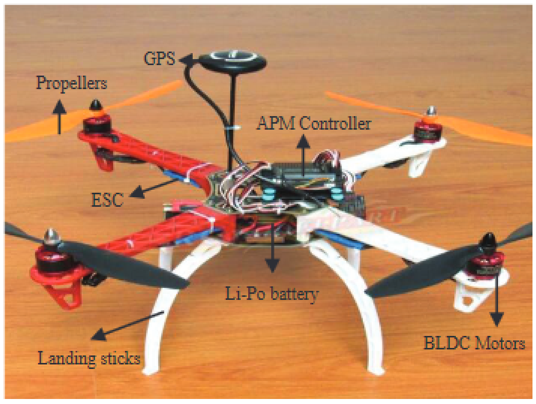


Figure 2.1: Example Drone Build [1]

There is no global standard for classification of UAVs. Depending on the functional context or the characteristics that the UAVs have, they can be classified into a myriad of categories.

For instance, the U.S. Department of Defense has issued a standardized classification for the UAVs used by the diverse branches of the U.S. military as follows: Micro, Mini, Tactical, Medium Altitude, and High Altitude UAV systems [9].

Even if there is no global standard for classification for each specific individual UAV, one can still separate UAVs into two main categories, fixed-wing and rotary-wing. Fixed-wing UAVs use their engine and propeller to generate forward thrust and their wings to manipulate the air pressure generated below them [10]. Due to the nature of fixed-wing aircrafts, this type of UAVs has to stay in a constant forward motion and, therefore, is unable to stay stationary while in flight. The ability to stay stationary while in flight is essential to many missions, which is why rotary-wing UAVs have been increasing in popularity and use. Rotary-wing UAVs manage to stay in the air due to their rotor based flight. The rotors present in the UAV supply it with propulsion and control, pushing the air down and providing it the ability to achieve vertical motion. By increasing or decreasing the thrust of the rotors one can achieve control over the vertical motions of the UAV. In order to achieve vertical motion without lateral motion, when the UAV has multiple rotors, all rotors must have the same increase or decrease of **rotations per minute (RPM)**, otherwise, the UAV will generate rotational force, **torque**. By being aware of the torque of a UAV and how to manipulate it, one can accurately drive in a lateral motion. The way a pilot can do this depends on how many rotors the UAV has. In the case of quadcopters, by changing the RPM of two rotors that are opposite to each other, the UAV will generate torque, rotating in one direction [11].

2.2 Dependability of UAV Systems

The dependability of a system is the main aspect that should be assured upon in order to avoid unwanted failure scenarios [12]. Dependability cannot be described as a single attribute, as there are several aspects that contribute to it. According to [13], dependability involves the following attributes:

- **Availability**: the systems and services should be mostly available, with very little or no down time;
- **Reliability**: the systems should behave as expected, with very few errors;
- **Safety**: the systems do not pose unacceptable risks to the environment or the health of users;
- **Confidentiality**: data and other information should not be divulged without intent and authorization;
- **Integrity**: data should not be modified without intention and authorization.
- **Maintainability**: maintainability of systems hardware and services should not be difficult or excessively expensive.

Software faults can be classified in physical and human-made faults [14]. In [15], we can find some widely accepted definitions for the two classes previously mentioned:

- **Physical faults**: adverse physical phenomena, either internal (physic-chemical disorders: threshold changes, short-circuits, open-circuits. . .) or external (environmental perturbations: electro-magnetic perturbations, temperature, vibration, etc.).

- **Human-made faults**: imperfections, defects, flaws, or bugs that do not occur but are instead introduced during the conceptualization, design, development (including maintenance), or configuration of systems. If not found, they will later manifest themselves as errors, which possibly turn into failures [13]. Human-made faults can be grouped in:
  - **Design faults**: committed either a) during the system initial design (broadly speaking, from requirement specification to implementation) or during subsequent modifications, or b) during the establishment of operating or maintenance procedures.
  - **Interactions faults**: inadvertent or deliberate violations of operating or maintenance procedures.

Hardware faults fall into the physical class and both are usually used interchangeably as terms. This is due to the lack of complex variety on what can be considered a physical fault. Software faults, however, although most commonly associated with design faults, can have a varying range of origins, and as such can be further cataloged into other classes. According to the Orthogonal defect classification (ODC) [16], software faults can be categorized into the following classes:

- **Function faults**: incorrect or missing implementation that requires a design change to correct the fault.
- **Algorithm faults**: incorrect or missing implementation that can be fixed without the need of design change, but this kind of faults requires a change of the algorithm.
- **Timing/serialization faults**: missing or incorrect serialization of shared resources. Certain mechanisms must be implemented to protect this kind of fault from happening, such as MUTEX used in operation systems.
- **Checking faults**: missing or incorrect validation of data, incorrect loop, or incorrect conditional statement.
- **Assignment faults**: values assigned incorrectly or not assigned.

Although the previously presented faults are the most prevalent designations of system faults that can be encountered, since the main objective of our work is to have realistic simulations and obtain results that can be applied to real scenarios, we mostly refer to failures to classify exploits and faulty states. Consequently, classifications like the Orthogonal defect classification are not frequently used in our work and are instead occasionally referenced to provide insight into the faulty state being presented. It is important to reinforce that while faults are flaws present in the system, only once these faults generate erroneous behavior that is propagated throughout the system and cause an unpredictable state do they turn into failures. Due to the immaturity of our research area, there is not an official failure classification for drones. Our classification is based on our research into the main exploitable aspects of a UAS and it is as follows:

- **Software Failure**: any failure caused by an internal fault in a software module.
- **Communication Failure**: any failure caused by the loss of communication within the UAS.
- **Artificial Intelligence Failure**: any failure caused by erroneous AI behavior.

- **Security Attack**: any failure caused by third-party attacks to any component of the UAS.

Some of the articles that mirror our choice on failure classification would be the article presented in [17] that evaluates the performance of UAV-based sensor networks under cyber-attacks, the research done in [18] which reinforces important issues in UAV communication networks and the wide variety of risk assessment works presented in section 2.4.

Every process that during the conceptualization or implementation of a system contributes to the previously mentioned attributes of dependability and, as a result, minimizes the number of existing faults can be considered a *fault avoidance* process/technique. Some examples would be standards, software quality assurance techniques/tools, and best practices for programming [19]. Even after considering fault avoidance processes and applying them in the most flawless way possible, the likelihood of bug-free software is very low due to the increase of software complexity in the latest years. Therefore, after trying to avoid as many faults as possible, one must think about how to tolerate those that will inevitably exist within a program.

With the increase in complexity of systems, *fault tolerance* quickly became one of the key aspects to take into consideration when writing software. Instead of trying to fully eliminate faults from the software, fault tolerance methods try to achieve dependability through having a system that is able to function, in normal conditions, alongside said faults. Since both fault avoidance and fault tolerance methods try to increase the dependability of a system, there is a considerable overlap of what these two methodologies aim for. Even though this is the case, the main aspect that we must consider in order to distinguish between the two is how they achieve their intended goal. While fault avoidance prevents the occurrence of faults by design and construction, fault tolerance provides the intended service in spite of faults occurring, for instance by redundancy [15].

In order to verify the tolerance of a system against faults, specific testing/evaluation techniques must be used. Among these techniques, fault injection stands out as one of the oldest ones, dating back to the early 70s [20]. *Fault injection* consists of either the purposeful exploitation of existing vulnerabilities or injecting faults through, for example, code mutation in a controlled environment. Through a controlled environment, one can frequently accurately and reliably determine fault coverage. This is important; as it allows developers and engineers to observe and evaluate the extent of damage said fault could have on the system while under normal working conditions.

Due to multiple factors such as higher cost systems and higher interest in fault tolerance, the industry had to adapt to provide effective and efficient fault injection tools. Hardware fault injection has become an inefficient way of testing fault tolerance for many systems, mostly due to the risk it carries of damaging real systems and the lack of need for testing at such a detailed level that this type of injection provides. Software fault injection does not carry the physical risks that hardware fault injection does, however it is more limited, as it is unable to inject faults into locations that are inaccessible to the software and is more oriented towards implementation details [21].

The fault injection types previously mentioned have both more advantages and disadvantages to them than the ones mentioned. The realm of robotics is the main focus of this work and in this context the factors mentioned are crucial. With the increase cost of development and the extensive testing a system should be subject to, damaging real systems could have severe consequences. As robots have both hardware and software components, in order to fully test them as a system it is not viable to only look at their implementation.

According to what was said previously, simulation-based fault injection is potentially the best alternative to test fault tolerance in robots, as it is possible to simulate the system while achieving maximum controllability and without intruding on the real system. As such, we believe this to be the most suitable fault injection type for our project. The quality of results when using this method is volatile, as it depends on how faithful the models used are, nevertheless it is still one of the most affordable and customizable types of fault injection [22]. Besides the ones mentioned, there are other types of fault injection, such as emulation-based or hybrid ones [23]. These tend to be considered more specific and experimental, mostly due to the lack of research when compared to hardware, software and simulation-based ones.

2.3 Safety of UAVs

To provide better automated services to the general population, ensuring the safety of UAVs operations is especially important. A remarkable characteristic of these operations is their scalability. The bigger the amount of drones present in said operations is, the higher the risk of failure and the potential damage that can be caused. The smooth integration of UAVs within civilian airspace relies on the ability to demonstrate that their operation would at least present a level of safety comparable to human-piloted aircraft.

In order to make this comparison possible, the work presented in [24] introduces an empirical analysis on safety objectives based on human-pilot aircraft activity. This analysis takes into consideration hazards present in both manned and unmanned aircrafts, and historical data of both on board fatality risks and ground fatality risks. In regard to on board fatality risks, the work states that within the analyzed data, 56% of midair collisions by human-piloted aircrafts were fatal but only 1% resulted in fatalities on the ground. They present a theory that states that although UAVs could exceed the midair collision accident rate by a significant margin, the number of fatalities would still be within an acceptable safety standard, due to the lack of an on board pilot. Regarding ground fatality, the article deduces that it is difficult to set an objective comparison between manned and unmanned vehicles. This is mostly due to the existence of UAVs that are physically incapable of causing fatalities due to their size (e.g. micro air vehicles). The article ends the first chapter by stating that fatality based analysis lead to certain limitations. Fatality as a metric does not reflect the occurrence of a hazard but only its consequence. The work gives the example of an accident that causes the death of eight people against eight individual accidents with each causing an individual death. Although the number of fatalities is the same, the higher risk rate present in the second case is not reflected in said number.

The article's chapter 2 addresses the impact that specific safety objectives would have on UAV operations and design. Concerning UAV operations, the authors establish a necessity for population density thresholds, above which UAVs would not be permitted to operate. It is also stated that although not directly violating safety objectives, operations at higher altitudes would generate a larger affected region for potential damage, making them more unpredictable. The impact on the design of UAVs systems is mostly addressed through illustrative examples but the authors distinctly mention that in order to increase the reliability of the systems, critical components must achieve a high level of redundancy and assurance.

2.4 Risk Analysis of UAVs

We decided to address a few select works that in some way target topics that are relevant to future chapters. Some of the topics and themes include, but are not limited to: UAV risk analysis applied to safety and/or security and fault injection on simulated or real UAVs. A summary of the articles referenced in this section can be seen in table 2.1.

Table 2.1: Article Summary for Section 2.4

Article	**Authors**	**General Objective**
Safety Risk Assessment for UAV operation [25]	Wackwitz, Kay Hendrick, Boedecker	Present a four-phase model of a UAS safety risk assessment
Qualitative and Quantitative Risk Analysis and Safety Assessment of Unmanned Aerial Vehicles Missions Over the Internet [26]	ALLOUCH, Azza, et al	Conduct detailed studies on UAV safety assessment at both qualitative and quantitative levels
Preliminary Risk Assessment for Small Unmanned Aircraft Systems [27]	BARR, Lawrence C. , et al	Present the development of two preliminary risk analysis approaches for small unmanned aircraft systems
Operational issues and assessment of risk for light UAVs [28]	G. Guglieri F. Quagliotti G. Ristorto	Provide a comprehensive insight for mission feasibility and operational implications in a set of a realistic applications cases
Threat and Risk Assessment Methodologies in the Automotive Domain [29]	Georg Macher Eric Armengaud Eugen Brenner Christian Kreiner	Examines threat and risk assessment techniques that are available for the automotive domain and presents an approach to classify cyber-security threats
The Vulnerability of UAVs to Cyber Attacks - An Approach to the Risk Assessment [30]	HARTMANN, Kim STEUP, Christoph	Demonstrate developed scheme for the risk assessment of UAVs based on provided services and communication infrastructures

We started by analyzing [25], which was written with the objective of providing UAS operators with guidelines which can be followed in order to write a documented safety risk assessment and a manual of operations, both of which will have to be presented in the future by certain pilots to the European Aviation Safety Agency (EASA). This article presents a four-phase model of a UAS safety risk assessment. These four phases are:

- Safety Hazard Identification
- Safety Risk Assessment
- Safety Risk Mitigation
- Safety Documentation

The methodologies presented for Hazard Identification are as follows:

- Reactive: Involves analysis of past outcomes or events. Hazards are identified through investigation of safety occurrences.
- Proactive: Involves analysis of existing or real-time situations during drone operation.
- Predictive: Data gathered is used to identify possible negative future outcomes or events during drone operation.

In regard to Risk Assessment, the article succinctly states that risks should be categorized in relation to their probability of happening, their severity in the case that they do happen and a joint evaluation of both previously mentioned characteristics. Methodologies for Risk Mitigation are also presented, namely:

- **Corrective actions**: Actions with an immediate effect for the safety hazard.
- **Preventive actions**: Actions that have a long-term effect on the safety hazard to mitigate the risk to an acceptable level.

The article [25] also demonstrates how to use the introduced guidelines through an illustrative use case. The article is also a good starting point if the reader wishes to learn more about the topic as it presents a well written summary on the main aspects that relate to safety risk assessment of UAVs.

Still on the topic of risk analysis, [26] provides a combination of qualitative and quantitative analysis to identify hazards and risks that can occur when drones are tele-operated over the internet. Their qualitative analysis makes use of the combination of two safety standards in order to identify system limits and hazardous conditions, along with their possible causes and their consequences. The functions that put the safety of the UAV in jeopardy are also identified and through them, a performance level – a value that dictates how high the performance of the safety related control needs to be – is calculated. Their quantitative analysis however, applies a method based on Bayesian networks with the objective of analyzing the relationships between the risk of drone crashes and their causes. Although this analysis makes use of collected civil UAV accidents and expected probabilities of occurrence of crashes associated with UAVs, the authors do not provide real use cases in their scenario analysis but instead use illustrative scenarios.

The authors of [27] follow a similar approach to the previous article. Their objective is to present the development of two preliminary risk analysis approaches for small UAS. These two approaches – a Standard Safety Risk Management Assessment and a Probabilistic Model-Based Risk Assessment – also operate at qualitative and quantitative levels. The first approach makes use of uses cases collected from the industry and an analysis on UASs mishaps in order to identify current hazards. The hazards are then subjected to a qualitative risk assessment based on operational complexity, population density, vehicle weight and configuration. After this, the hazards are then tabulated according to their severity and likelihood, and both tables are added in a risk matrix. Using this matrix, a determination of the probability of an errant quadrotor striking a person on the ground, as a result of loss of control caused by flight control system component failures, is then done in five operational environments: remote, rural, suburban, urban and congested. In their second approach, they make use of a Bayesian Belief approach in order to model a UAS operational environment. Through the application of an object-oriented capability software, the authors populate their model and provide a test scenario visualization through the simulation of a loss of control, which propagates a failure state throughout the simulated UAV's propulsion and navigation systems.

Although there are certain risk assessment models that are more widely used, there is a lack of an accepted objective model throughout the research community. This is mostly due to the difficulty in providing a general model, given the amount of different scenarios an UAV can go through and the risks that can originate from such scenarios. The authors of [28] present two different methods for risk assessment that, for the most part, focus on the damage caused by the drone in the case that a failure happens. Meanwhile, [29] presents a new method, derivative from two already existing approaches, that is aimed towards the needs of an analysis of security threats during conceptualization phases. The work presented in [30] demonstrates a developed scheme for risk assessment based on provided services and communication infrastructures. They aim to evaluate how environmental, communication links, sensors, data storage and fault handling mechanisms risks affect a set of UAVs in regard to integrity, confidentiality and availability.

2.5 Safety Assessment of UAVs

In regards to UAV exploitation, the author of [31] performs a series of planned attacks – DoS, Wireless Injection and Video Capture Hijack – to a physical drone in order to show the effect of miniaturization on a commercial drone. The purpose of the work is to show the ease with which one can remotely access a drone and as such, the author does not write about the UAVs software in detail nor presents a representative fault model.

In [32], the authors present, in a succinct way, a multi-layered security framework that can potentially be implemented in order to protect drones against specific vulnerabilities. It is verified that the vulnerabilities explored – through buffer-overflow attack, DoS and Address Resolution Protocol Cache Poison attack – are, in fact, present in the commercial UAV used. As a complement to the framework, the work also presents three algorithms that can, in theory, mitigate the effects that the explored vulnerabilities can have on the UAV. Unfortunately, these algorithms are not implemented and demonstrated in a practical setting and, as such, no comparison is made between the UAV's performance while on faulty behavior versus the UAV equipped with the proposed framework.

The authors of [33] aim to demonstrate through a security assessment – namely, a De-Authentication attack - that anyone with access to a computer could potentially take down a drone. They also dedicate a chapter to discuss some prevention methods for the aforementioned attack. After detailing the way in which the attack was performed, the work evaluates the vulnerabilities found using the stride and dread models. As with the previous work by Hooper, Michael, et al, although the authors share ways in how prevention methods could be implemented, without the implementation of such methods their efficiency is not validated.

As with the previous articles regarding exploitation, the authors of [34] aim to assess the security of commercially available drones. They present four security vulnerabilities found in two drones and exploit them through a series of attacks – De-Authentication attack, File Transfer Protocol Service Attack, Radio Frequency Replay Attack and a Custom Made Controller Attack. The work has a chapter dedicated to countermeasures but lacks a demonstration of said countermeasures, and as such, their efficiency is not proven.

The four previous works focus on performing exploitation tactics on a real UAV, in [35] however, the authors present a fault detection architecture used to enhance analysis capabilities for critical safety properties and reduce costs of related UAV systems using Hardware in the Loop (HITL) simulation. They make use of a fault injection manager to inject faults that are carried for yaw, pitch, roll and wind speed parameters. HITL simulation is an useful compromise, as it allows to use real systems in simulated scenarios. This method is still limited by the available drones, but presents less risks than experiments under real conditions.

The work presented in [36] varies from the ones previously shown as its experiments are performed in a fully simulated environment. The authors analyze the effects of GPS spoofing effect on UAVs through a series of tests. Their Software in the Loop simulator models all necessary hardware and allows them to alter values such as noise in sensors or wind speed. The trajectory used in the experiments which consists of a takeoff, a two lap horizontal line and a landing, could be harmful if done in a real environment. However, in a simulated environment even in the event of an unpredictable failure, no harm can be caused. The authors of [37] also make use of a simulated environment and propose the use of a support vector machine classification algorithm for the fault diagnosis of an octorotor after simultaneous or successive motors failures. The implementation of the algorithm is accompanied by its validation through the presentation of two scenarios. The scenarios are based on simulating motors failures on a real octorotor UAV by using fault injection.

Whilst it is not focused on safety but on security risk assessment, the authors of [38] focus on exploring terrestrial to aerial communication and securing it. They present countermeasures to protect communication from eavesdropping and jamming, and provide numerical results in order to demonstrate the effectiveness of some of said countermeasures.

We believe that the inherent limitations that a real system has are reflected on the lack of variety of exploitation tactics found in the literature. Although we presented works in which there are instances of simulation based testing being used, we consider that there is still a lack of works that provide a throughout insight on safety risk assessment and results validation, applied to a simulated environment. In table 2.2, a summary of the most relevant articles mentioned in this section is shown.

Table 2.2: Article Summary for Section 2.5

Article	**Environment Used**	**Exploitation method**	**Failure Type**
ARDrone corruption [31]	Real UAV scenario	DoS, Wireless Injection and Video Capture Hijack	Securiy Attack
Securing commercial wifi based uavs from common security attacks[32]	Real UAV scenario	DoS, buffer-overflow attack and ARP Cache Poison Attack	Security Attack
A Security Assessment for Consumer WiFi Drones [33]	Real UAV scenario	De-Authentication attack	Security Attack
Assessing and exploiting security vulnerabilities of unmanned aerial vehicles [34]	Real UAV scenario	De-Authentication attack, FTP Service Attack, RF Replay Attack and a Custom Made Controller Attack	Security Attack
Fault tolerance system for UAV using hardware in the loop simulation [35]	HITL simulation	Inject faults that are carried for yaw, pitch, roll and wind speed parameters	Software Failure
Effects of GPS spoofing on unmanned aerial vehicles [36]	SITL simulation	GPS Spoofing	Security Attack
Fault Diagnosis and Fault Tolerant Control of an Octorotor UAV using motors speeds measurements [37]	SITL simulation	Inject faults that recreate motor failures	Software Failure

Chapter 3

Objectives and Approach

This chapter addresses the objectives determined for the duration of our work and the approach taken for each individual goal.

3.1 Objectives

As was briefly mentioned in Chapter 1, **the principal objective of this work is the design and implementation of a fault injection technique and tool that can be used to assess the safety aspect of diverse types of autonomous quadcopters.** In order to achieve this objective, some groundwork is necessary and several goals needed to be reached, namely:

- Definition and characterization of the SUA and its environment (i.e., context and scope of the work);
- Identification and characterization of failure scenarios;
- Creation of a representative and possibly complete fault model (i.e., including both generic and SUA's specific faults);

The built environment must be able to fully recreate a working autonomous quadrotor and all the components associated with it. To fulfil this, a research on the available tools to recreate UAV behaviour was done. Due to a lack of detailed works on individual software, an objective choice on which software was the most appropriate was sometimes not possible. In these cases, after some research, we concluded that several options were valid as they aimed to achieve the same goal through different methods. During the fault identification phase it was taken into consideration that this work mainly focus on safety assessment and as such, the shown faults are not representative of all the identified faults but instead of all the faults we believe to be relevant to the topic at hand. The representative fault model presents the identified faults in a concise manner, providing direct characterization and descriptions.

We aimed to implement the faults present in the fault model (Table 5.5), and implement a tool that manages their presence within the code. We had also intended to perform a risk analysis of said faults, but later discarded this objective, as we believed it to not be in accordance with the practical nature of our work.

Depending on the complexity implementing the exploits may have, it may not be possible to implement all of them. Once the tool is implemented, we intend to perform experiments in a more organized and efficient way. The experiments will relate to the assessment of the system, and will be planned and executed during the end of our work's cycle.

3.2 Research Approach

Our initial objective was the identification of all components involved in an autonomous quadrotor and building a representative model that could recreate the functionalities of said components. The identification of all the components was accomplished throughout the first phase of our work and led to the writing of section 2.1. With the intent of building an accurate model, we performed a research on the available software for the simulation of quadrotors as shown in section 5.1. We started by choosing a simulator that had the capacity of physically representing a drone. As a simulator is only capable of representing a drone but not provide its software capabilities, we had to select a flight control software that could function in parallel with our simulator. A more detailed explanation on each software can be found in sections 5.2.1 and 5.2.2.

After building the representative system we had to create a fault model for it. Generally, the fault model is a concise presentation of faults that represent the applied fault injection method. With safety assessment in mind, we analyzed the various software modules of our flight control software with the purpose of finding instances where the software could be exploited. The results of the initial analysis and the developed fault model are explained in detail in section 5.3.

Following this, we had planned to perform a risk analysis on the identified faults by applying the risk assessment model, the DREAD model. As mentioned in the previous section, we later found that this objective did not correspond to the practical nature intended for our work.

As such, the step that followed the development of the fault model, was the conceptualization of our fault injection tool. We accomplished this by making use of a web-based prototyping tool, **Figma**.

Once the conceptualization phase was over, we then started the implementation phase. During this phase we simultaneously worked on the implementation of faults - mainly GPS faults - and the implementation of the previously conceptualized interface. The tool was built using the integrated development environment (IDE) **Qt Creator**, which allowed us to efficiently recreate the imagined interface.

After implementing the tool and a variety of faults, we performed an array of experiments to validate the implemented elements. The results gathered from these experiments were then analyzed, so that a conclusion regarding the effectiveness of the tool could be drawn. A representation of the research approach taken is reflected in image 3.1. The objectives encapsulated in a blue box were accomplished during the first phase of our work, while the ones encapsulated in an orange box were accomplished during the second phase.

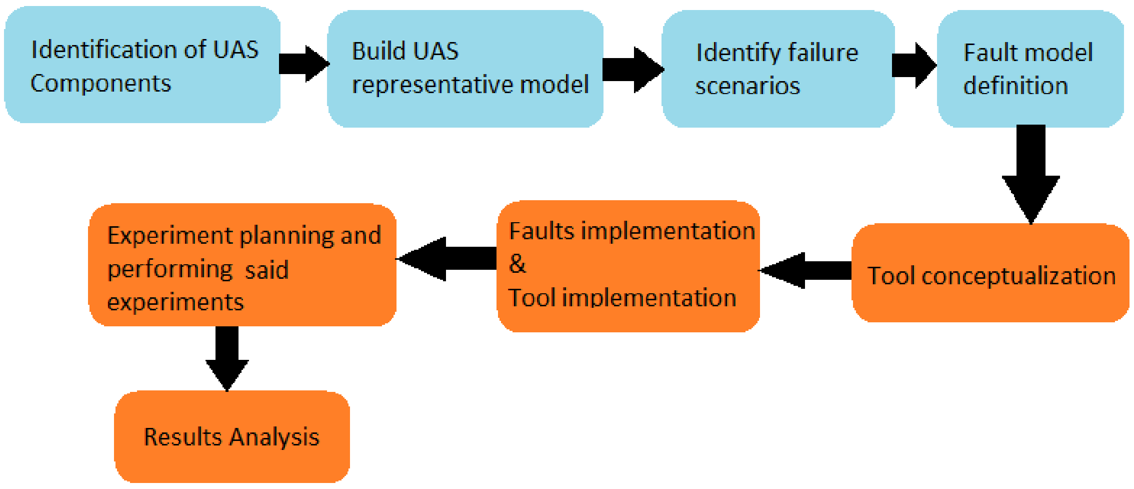

Identification of UAS Components
Build UAS representative model
Identify failure scenarios
Fault model definition
Tool conceptualization
Faults implementation & Tool implementation
Experiment planning and performing said experiments
Results Analysis

Figure 3.1: Representation of the planned research approach

Chapter 4

Context

This work is being developed as an offshoot of project BUBBLES, which is a project targeting the formulation and validation of a concept of separation management for UAS in the U-space.

The U-space can be described as a safe and secure airspace that is designed to facilitate any kind of routine mission for a large numbers of drones. To achieve this goal, new services and specific procedures will need to be properly researched and proven. A generic representation of the U-space can be seen in figure 4.1. The represented U-space station can provide a series of services in order to enable safe operations between the UAVs present in the U-space. These services can be divided between a **strategic phase** and a **tactical phase**. While the former provides services related to the creation and authorization of flight plans for each individual UAV, the latter provides services that monitor the current state of the U-space and that simulate future states. Both phases are respectively represented in figures 4.2 and 4.3.

Currently the BUBBLES programme is divided into 8 work packages:

- WP1 - Project management & coordination
- WP2 - Concept definition & refinement
- WP3 - ConOps definition & classification
- WP4 - Separation minima & methods
- WP5 - Concept validation
- WP6 - Safety, Performance and Interoperability
- WP7 - Performance monitoring
- WP8 - Communication, dissemination & exploitation

The relations between these packages is depicted in figure 4.4. Our work acts as a supplement to work package 5, by providing the BUBBLES project with UAS flight trials in a controlled research environment in which the fault occurrence is accelerated using fault injection. As such, we aim to provide a detailed planning of the experiments for testing the tool and accurate tests, in order to supplement the research done in regard to the evaluation of the impact of the different types of faults that may affect the different elements of the system. This experimental assessment will allow a realistic definition of failure parameters and fault impact, which will be used in BUBBLES in the models for risk assessment and safety evaluation. To accomplish this, we scheduled meetings with important members of the project in order to better adapt to its needs.

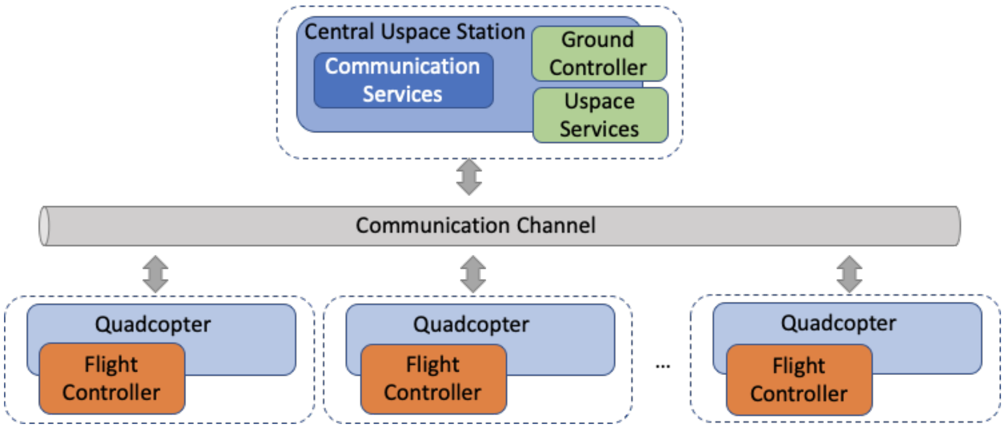


Figure 4.1: Generic Representation of a U-space environment

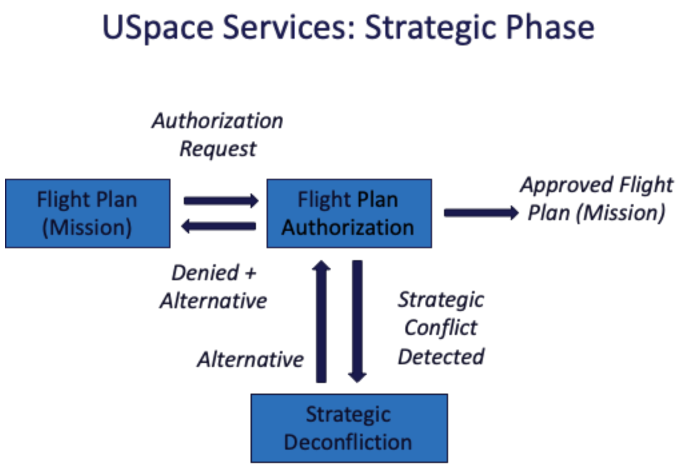

USpace Services: Strategic Phase
Authorization Request
Flight Plan (Mission)
Flight Plan Authorization
Approved Flight Plan (Mission)
Denied + Alternative
Alternative
Strategic Conflict Detected
Strategic Deconfliction

Figure 4.2: U-space services: Strategic Phase

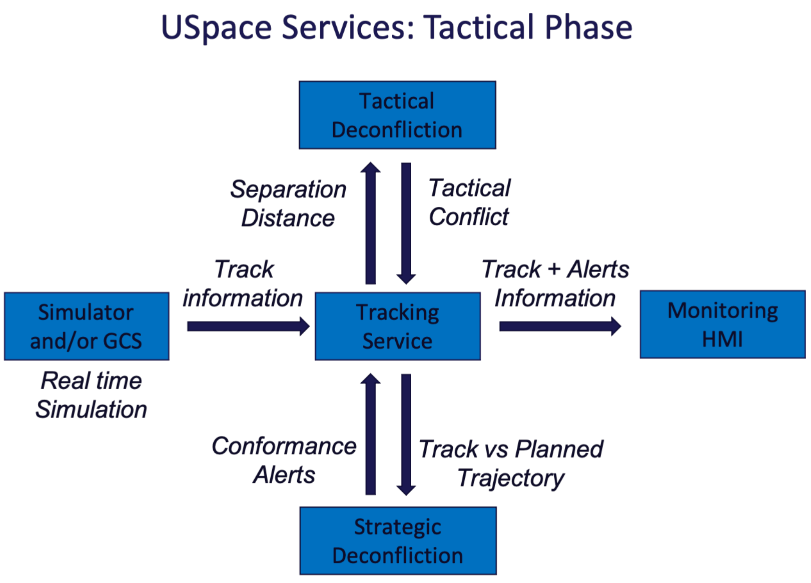


Figure 4.3: U-space services: Tactical phase

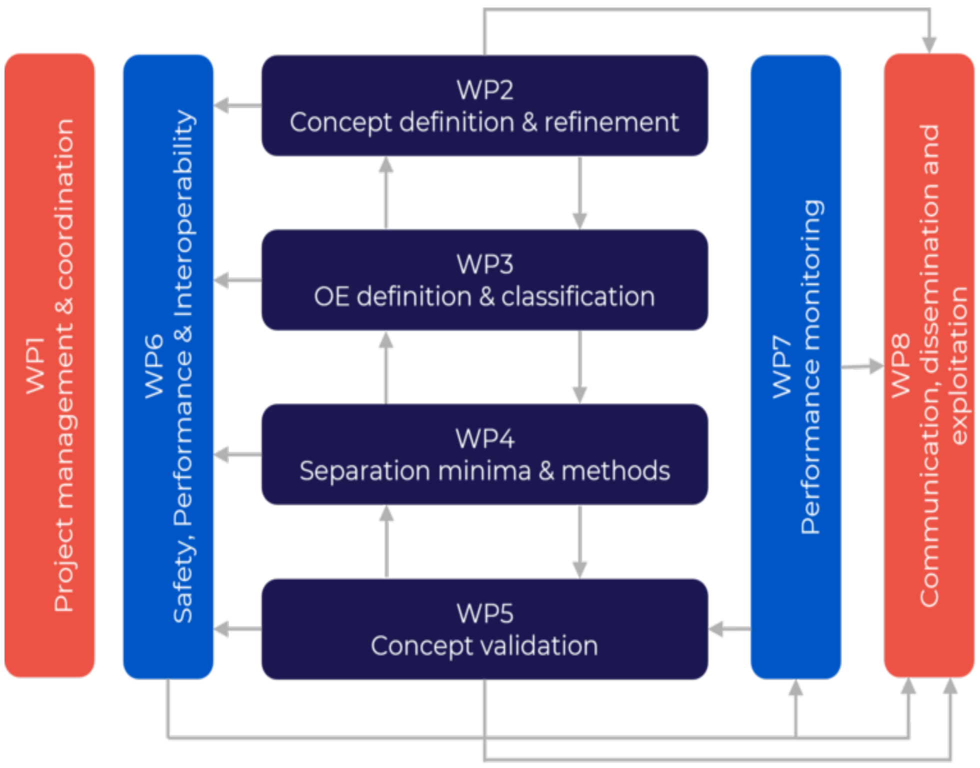

WP1
Project management & coordination
WP6
Safety, Performance & Interoperability
WP2
Concept definition & refinement
WP3
OE definition & classification
WP4
Separation minima & methods
WP5
Concept validation
WP7
Performance monitoring
WP8
Communication, dissemination and exploitation

Figure 4.4: BUBBLES's 8 Work Packages

Chapter 5

System under Assessment

This chapter will present the preparation of the system under assessment and its environment, as well as a compilation of viable research exploits in the form of a fault model.

5.1 Preparation of the SUA and its Environment

In this section, we go through some of the various options available to date that allow the assembly of simulation environments for drones, as well as the flight control software chosen for our intended purposes.

Robot Operating System, shortened as ROS, is an important tool in robot software development, as it supplies developers with a wide array of tools for automating vehicle control and data management. Therefore, although not a hard requirement when wanting a functional simulator, for research and development purposes, ROS compatibility is a must. Luckily, due to the open-source nature of many of these simulators, nowadays, most of them are able to work alongside ROS either due to official updates, or community contributions.

Table 5.1: Simulators analyzed

Simulator	**Purpose**	**Supports ROS**	**System requirements**	**Supported Drone Types**	**Supports multiple drones**
ARGoS	Large-scaled Swarms Research and Development	Yes	Lightweight	Small sized wheeled	Yes
Webots	Validating AI Algorithms Robotics Educational Tool	Yes	Mediumweight	Wheeled and humanoid	Yes
CoppeliaSim	Motion Planning Robotics Educational Tool Virtual Reality Drone Research	Yes	Mediumweight	Wheeled, humanoid and flying	Yes
Gazebo	Drones and Sensors Research done on Realistic Environments	Yes	Mediumweight	Wheeled and flying	Yes
jMAVSim	Accessible Simulator aimed at assessing Quadcopter Behaviour	Yes	Lightweight	Flying	Yes
AirSim	AI Algorithms Research on highly visually Realistic Simulations	Yes	Hardweight	Flying	Yes

Since we required our simulator to simulate quadcopter behavior, CoppeliaSim, Gazebo, jMAVSim and AirSim are the ones that accomplish this initial requirement. We decided to cut AirSim due to its steeper system requirements and jMAVSim due to its simplicity and lack of features. The choice between Gazebo and CoppeliaSim will be dependent on the compatibility both have with the available selection of flight control software.

Although an environment with just a simulator and ROS is possible, depending on the libraries used, in order to build the most accurate and reliable environment, we decided to use a flight control software that will be active alongside ROS. The flight control software component can be considered an extension of the functions provided by ROS, to the point that some of the presented options are packages found within ROS. This specific type of software provides certain UAV models with advanced routines and functionalities, which in turn allows the simulated UAVs to further replicate real behaviour. Our goal was to choose a software that supports both ROS and either Gazebo or CoppeliaSim.

Table 5.2: Flight Control Software analyzed

Flight Control Software	**ROS Compatible**	**Gazebo Compatible**	**CoppeliaSim Compatible**
PX4	Yes	Yes	No
Ardupilot	Yes	Yes	No
Rotors	Yes	Yes	No
Hector Quadrotor	Yes	Yes	No

Many of these software have already reached a state where they can all achieve, generally speaking, the same tasks. As such, many of the arguments that can be made in order to favour one over another, do not mention the features of the software itself, and instead focus on licensing, ease of use, hardware requirements, or other aspects not related to the capabilities of the software. Hector and Rotors are considered outdated when compared to PX4 or Ardupilot as development has ended a few years ago and in general, are not considered good options for simulations with the intent of research and real testing due to the lack of features. For that purpose, it is recommended to use packages interfacing stacks such as PX4 or Ardupilot, as those are the software used in most physical drones. Choosing between PX4 and Ardupilot was a tough choice, especially in a simulated environment, as a lot of arguments used are related to real world uses. We ended up choosing PX4 for our environment, but both options aligned with our requirements and intended goals.

We also used the ground station software, QGroundControl, in order to supplement our environment. A ground station is typically a software application that can communicate with a desired set of UAVs via wireless telemetry. It displays real-time data on the UAVs performance and position allowing the user to be aware of this information in a more succinct way. Although we initially intended to use this software to upload missions and specific behaviours to our simulated drones, we later discovered that QGroundControl's API was severely lacking and as such, could not be used from within our tool. We still used this software to validate our fault implementation while the tool was not finished, but its initial role ended up being fulfilled by DroneKit.

DroneKit allows developers to create diverse apps that can be ran in conjunction with a drone. These apps can enhance a wide variety of the drone's characteristics (e.g. autopilot). It can also be used as a ground station app, which is our main interest. DroneKit is compatible with vehicles that communicate using the MAVLink protocol - this includes the vehicles used by our flight controller. It is also a cross-platform application, a highly valuable characteristic for us. [39]

Additional information regarding simulators and ground station software can found in [40] and [41]. Although it is not possible to dictate a clear superior simulator due to the wide array of objectives and to a substantial amount of overlapping capabilities that each simulator presents, both articles present a wider variety of simulators than the one presented in table 5.1 and briefly explain their purposes and features.

5.2 Simulated Environment

In this section the various choices in regard to the chosen software are presented and explained. We also discuss some illustrative representations of the SUA.

5.2.1 Flight Control Software

As previously stated, our environment will be based on PX4, an open-source autopilot system oriented towards inexpensive and accessible autonomous aircraft. PX4 is physically based on two main layers: the flight stack, which mainly functions as a flight control system, and the middleware, a general robotics layer that can support a multitude of autonomous robots, providing internal/external communications and hardware integration.

The diagram shown below (Figure 5.1) provides a detailed overview of the building blocks of PX4. The top part of the diagram contains middleware blocks, while the lower section shows the components of the flight stack.

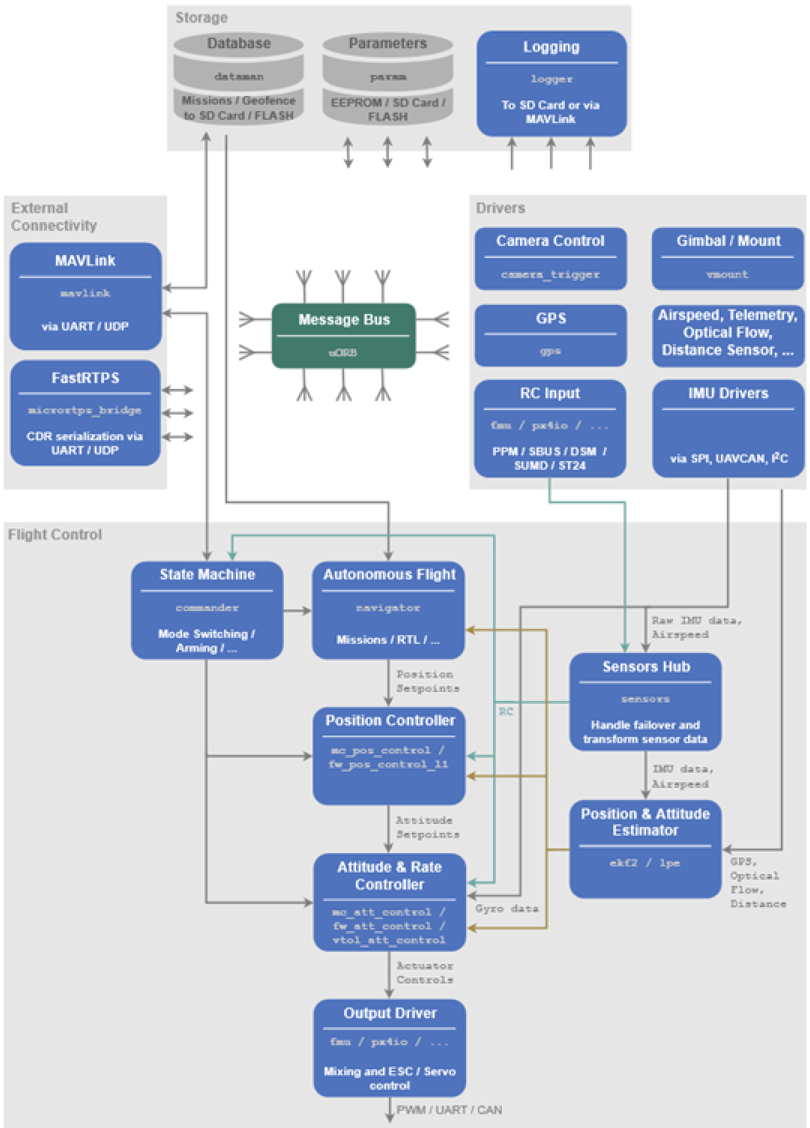


Figure 5.1: PX4 High-Level Software Architecture Diagram [2]

The actual number of connections between the modules is not portrayed accurately, as there are many more. Only the most relevant ones are demonstrated in the diagram, as portraying every connection would lessen the diagram's readability. In truth, some data can be accessed from most of the modules.

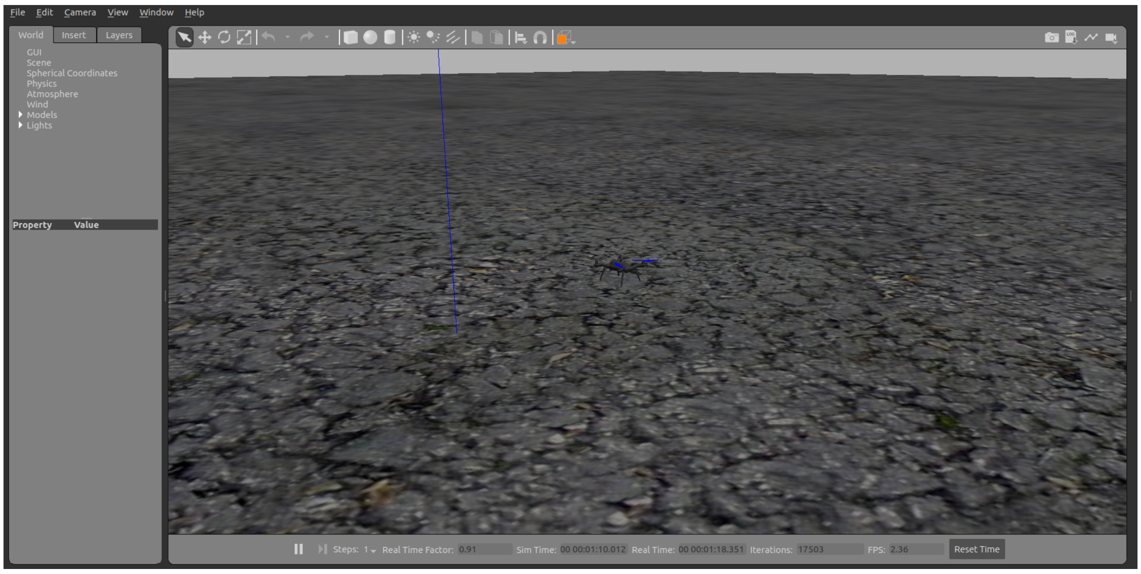
Modules communicate with each other through a publish-subscribe message bus named *uORB*. The use of the publish-subscribe scheme means that [2] :

- The system is reactive — it is asynchronous and will update instantly when new data is available
- All operations and communication are fully able to happen in parallel
- A system component can consume data from anywhere in a thread-safe fashion

5.2.2 Simulator

As mentioned in section 5.1, for our simulation software we chose Gazebo, which is an open-source 3D robotics simulator. Gazebo's rise in usage and popularity was due to an increase in the usage of robotic vehicles with outdoor oriented goals. As such, it was developed with the intent of giving the user the ability to accurately reproduce environments that may be traversed by said robots. Gazebo excels at portraying complex bodies through a combination of simple geometric shapes and joints. These joints connect the shapes together allowing them to perform dynamic motions [42].

A focal point for our work is Gazebo's wide library of sensors. In order to accurately simulate a quadrotor, being able to replicate the wide array of information that the drone needs is essential. The simulator also provides its users with the ability to create their own models and share them publicly. This allows the community to rapidly increase the public repertoire of available robots.



Steps: 1 Real Time Factor: 0.91 Sim Time: 00 00:01:10.012 Real Time: 00 00:01:18.351 Iterations: 17503 FPS: 2.36

Figure 5.2: Drone flying in a basic world in Gazebo

In [43] the authors present a system that integrates both ROS and the Gazebo simulator. They also simulate different scenarios, mainly based on the flight of a simulated quadrotor. Through the use of various sensors - Inertial Measurement Unit, Barometric Sensor, Ultrasonic Sensor, Magnetic Field Sensor and GPS Receiver - and the implementation of a dynamics model based on flight dynamics, motor dynamics and thrust calculation, the simulator is able to accurately replicate a real UAV, as proved by validation experiments performed in the article.

5.2.3 High-level environment representations

Figures 5.3 and 5.4 succinctly represent at a higher level the relation between the various elements of the environment. Figure 5.3, in specific, demonstrates the possibility of having a N number of quadcopters being simulated in parallel. Through *mavros*, a ROS package, it is also possible for these quadcopters to be simulated in different computers, as long as the proper communication channel is set up.

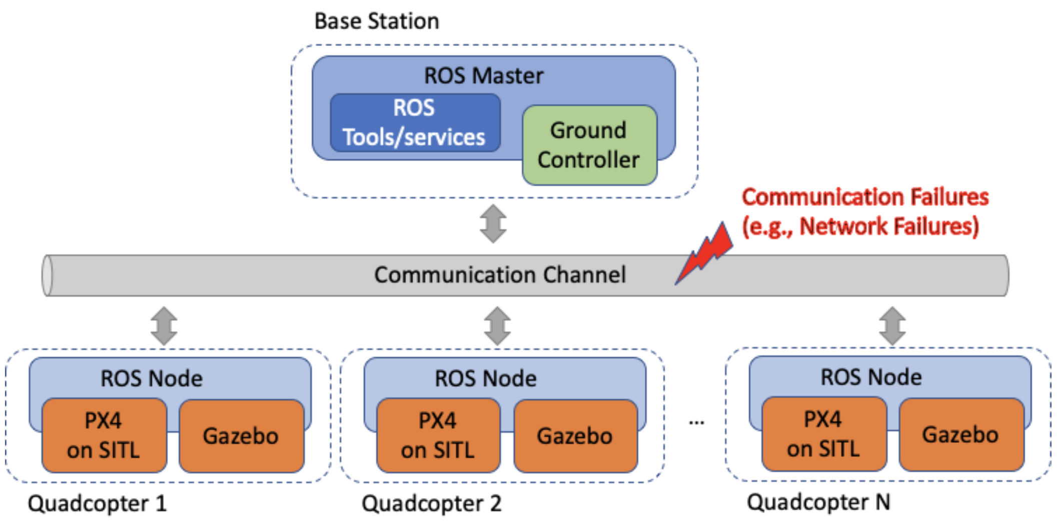


Figure 5.3: High Level Representation of the Communication between N Quadcopters

In figure 5.4 a more visual representation of the elements necessary to simulate a single quadcopter is shown. It is possible to observe the full PX4 Architecture Diagram, as shown in figure 5.1, as well as the most relevant elements of Gazebo simulator and a ROS communication Node similar to the ones shown in figure 5.3. In the Gazebo section, it is possible to observe elements such as:

- **Models for the UAV, the environment and the sensors**, these allows us to accurately simulate these elements;
- **The positioning system**, which allows us to portray the notion of coordinates and distance;
- **The actuator plugins**, which receive commands from PX4 based on the processed sensory data and applies them to the simulated actuator components.

Within the figure we also point out an array of faults that might be used to exploit the system. Their categorization is as follows:

- **Communication faults**, faults that target the MAVLink communication channel between PX4's MAVLink module and its respective ROS node.

- **Command faults**, faults that target commands issued by the State Machine module (e.g. abort command).
- **Environmental/Sensors/PS faults**, faults that target the data gathered by the sensors and corrupt it.
- **Mission faults**, faults that target the mission on cache whilst it is being read, possibly affecting trajectory.
- **Sensors/PS data transformation faults**, faults that target the data gathered by the sensors whilst it is being processed.
- **Estimator faults**, faults that target the processed data once it is sent to the modules that control the drone's movement.

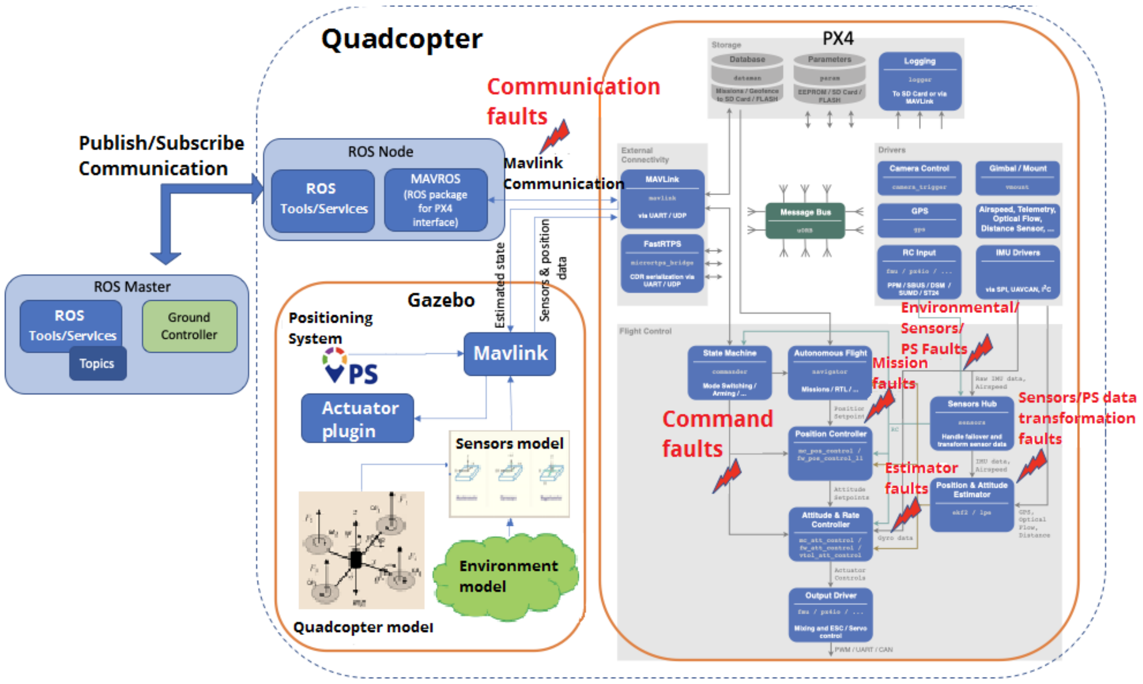


Figure 5.4: High Level Diagram of our Simulated Environment

5.3 Fault Model

In order to create our fault model we started by analyzing each software module represented in figure 5.1 individually. As our focus relates to safety assessment, for this initial approach we already had a preconceived idea of which type of exploits we wanted. As such, the shown exploit possibilities are not all the ones that were found, but instead all the ones that we believe adhere to our topic of research.

This analysis allowed us to select an initial batch of instances which we believe might wield interesting results through fault exploitation. In order to facilitate understanding, the batch is divided in two tables, 5.3 and 5.4. The former states in which module of the PX4 architecture each instance of possible exploitation can be found, within said module in which file, a brief description of the exploit's aim, as well as a classification of the exploitation using the failure types we present in section 2.2. The latter gives a more practical insight on each exploit stating how to cause the exploit, which variable within the file respectively specified in table 5.3 is mainly responsible for carrying pertinent data, the range of said variable and what's the end purpose of the function we are altering.

Together, the tables have 9 different columns with each representing a specific point that we found relevant to best summarize each exploitable instance. In a summarized manner, the meaning of each column is as follows:

- **ID** - a numerical attribute given in order to better identify each case;
- **Module** - the PX4 software module that will need to be modified;
- **File name** - the file, inside the module, that will need to be modified;
- **What can we achieve** - the intended goal of this exploit;
- **How do we achieve it** - simplified description of which values to alter inside the file;
- **Type of Problem** - type of failure that the exploit means to represent, this can be either a Software failure, a Communication Failure, an Artificial Intelligence Failure or a Sensor Spoof;
- **Variable to Alter** - which variable carries the most relevant data in accordance to our goal, more code besides this one variable will need to change, but this one is of most relevance;
- **Range of variables** - which range of values can the variable mentioned previously take;
- **What does the function return** - purpose of the function that will be modified and what does it send to other module, either as a value or as a message.

In some cases the column **Type of Problem** presents more than one type of failure per case. This is due to the possibility of recreating the same failure outcome through a different type of exploit.

By making use of the information demonstrated in tables 5.3 and 5.4, we were able to formulate our fault model, as shown in table 5.5. The fault model summarizes the definition of each planned fault injection by presenting a brief description of said fault, as well as classifying them using Orthogonal defect classification and our failure type classification presented in section 2.2.

Table 5.3: Information on the exploitable instances

ID	Module	File name	What can we achieve	Type of Problem
1	Camera Control	Camera_trigger.cpp	Modity distance detection capability	Software failure or Sensor Spoof
2	GPS	GPS.cpp	Report wrong GPS values to device	Software failure or Sensor Spoof
3	Sensors Hub	Vehicleacceleration.cpp	Alter published vehicle acceleration	Software failure
4	Sensors Hub	Vehicleairdata.cpp	Alter published barometer values	Software failure
5	Sensors Hub	Vehicleangularvelocity.cpp	Alter published vehicle angular acceleration	Software failure
6	Sensors Hub	Vehiclegpsposition.cpp	Alter published gps data	Software failure
7	Sensors Hub	Vehicleimu.cpp	Alter published inertia values	Software failure
8	Sensors Hub	Vehiclemagnetometer.cpp	Alter published magnetometer values	Software failure
9	Position & Attitude Estimator	EKF2Selector.cpp	Alter published vehicle global and local position	Software failure
10	Position & Attitude Estimator	EKF2.cpp	Alter published attitude, odometry and wind estimate values	Software failure
11	Attitude Control	Mc_att_control.cpp	Alter published rate setpoints	Software failure
12	Autonomous Flight	Geofence.cpp	Make it so the system does not know it is inside a geofence	Software failure or Communication Failure
13	Autonomous Flight	Navigator_main.cpp	Purposefully generate scenarios where traffic is incorrectly verified	Software Failure or Artificial Inteligence Failure
14	State Machine	Calibration_routines.cpp	Return incorrect drone orientation (when drone is still)	Software failure or Communication Failure
15	State Machine	Preflightcheck.cpp	Initiate takeoff even with module failures	Software failure or Communication Failure

Table 5.4: Practical information on the exploitable instances

ID	How do we achieve it	Variable to Alter	Range of Variable	What does the function return
1	Alter distance values at update_distance function	current_position	2D Vector limited by the world's maximum coordinates	No hard values but changes last_shoot_position to an improper value
2	Alter _report_gps_pos values at run() function	report_gps_pos	Range of float variable but soft limit of: Latitude: -90 to 90 | Longitude: -180 to 180	Publishes GPS values
3	Alter v_acceleration value at run() function	v_acceleration	3D Vector with range of float variable, since vehicle acceleration is related to physical capabilties, soft limit related to hardware	Publish v_acceleration value
4	Alter varied barometer values at run()	out.rho	Range of float variable, soft limit equal to maximum atmospheric pressure value on earth	Publish atmospheric pressure
5	Alter v_angular_acceleration value at run() function	v_angular_acceleration and v_angular_velocity	3D Vector with range of float variable, since vehicle acceleration is related to physical capabilties, soft limit related to hardware	Publish v_angular_acceleration and v_angular_velocity
6	Alter gps data at publish() function	gps_output	Range of float variable but soft limit of: Latitude: -90 to 90 | Longitude: -180 to 180	Publish GPS values
7	Alter data at run() function	imu	PX4 struct that contains multiple attributes	Publish inertia measure
8	Alter magnetometer_data value at publish() fuction	out.magnetometer_ga	3D Vector with range of float variable, soft limit related to hardware	Publish magnetometer data
9	Alter values at their respective publish functions	local_position & global_position	PX4 struct that contain public attributes related to the vehicle https://px4.github.io/Firmware-Doxygen/d4 /d36/structvehicle__local__position__s.html	Publish positions
10	Alter data values at their respective publish functions	att.q & odom & wind_estimate	att.q: Takes a quaternion odom: Range of float for position and linear velocity | Takes a quaternion wind_estimate: 2D Vector with range of float variable for wind velocity	Publish attitude, odometry and wind estimation data
11	Alter rate setpoints values at run() function	v_rates_sp	3D Vector with range of float variable roll, pitch and yaw values soft limited by hardware	Publish rate setpoint for roll, pitch and yaw
12	Alter the checkAll() function in order to set up wrong geofence infraction return values	inside_fence	inside_fence: True or False	Returns a bool; false means geofence violation
13	By altering the check traffic function, it is possible to make it so the drones ignore possible collision scenarios	vcmd.command	Enum type variable that can receive a range of commands https://px4.github.io/Firmware-Doxygen/d7 /d94/vehicle__command_8h_source.html	Publish commands to change drone behaviour
14	Alter detect_orientation function in order to return the wrong orientation	return value (enum variable)	Return value: ORIENTATION_ERROR ; ORIENTATION_TAIL_DOWN ; ORIENTATION_NOSE_DOWN ; ORIENTATION_LEFT ; ORIENTATION_RIGHT ; ORIENTATION_UPSIDE_DOWN ; ORIENTATION_RIGHTSIDE_UP	Enum variable value that dictates drone orientation
15	Alter preflight check function in order to ignore possible failures	failed	Failed: True or False	Returns preflight check report as a boolean

Table 5.5: Fault Model

ID	Fault Title	Fault Class	Failure Type
1	Send wrong distance values to the device	N/A	Security Attack
2	Send wrong gps data to the device	N/A	Security Attack
3	Alter published vehicle acceleration	Assignment fault	Software failure
4	Alter published barometer values	Assignment fault	Software failure
5	Alter published vehicle angular acceleration	Assignment fault	Software failure
6	Alter published gps data	Assignment fault	Software failure
7	Alter published inertia values	Assignment fault	Software failure
8	Alter published magnetometer values	Assignment fault	Software failure
9	Alter published vehicle global and local position	Assignment fault	Software failure
10	Alter published attitude, odometry and wind estimate values	Assignment fault	Software failure
11	Alter published rate setpoints	Assignment fault	Software failure
12	Make it so the system does not know it is inside a geofence	Checking fault	Software failure
13	Purposefully generate scenarios where traffic is incorrectly verified	N/A	Artificial Intelligence Failure
14	Return incorrect drone orientation (when drone is still)	Checking fault	Software failure
15	Try to initiate takeoff even with module failures	Checking fault	Software failure

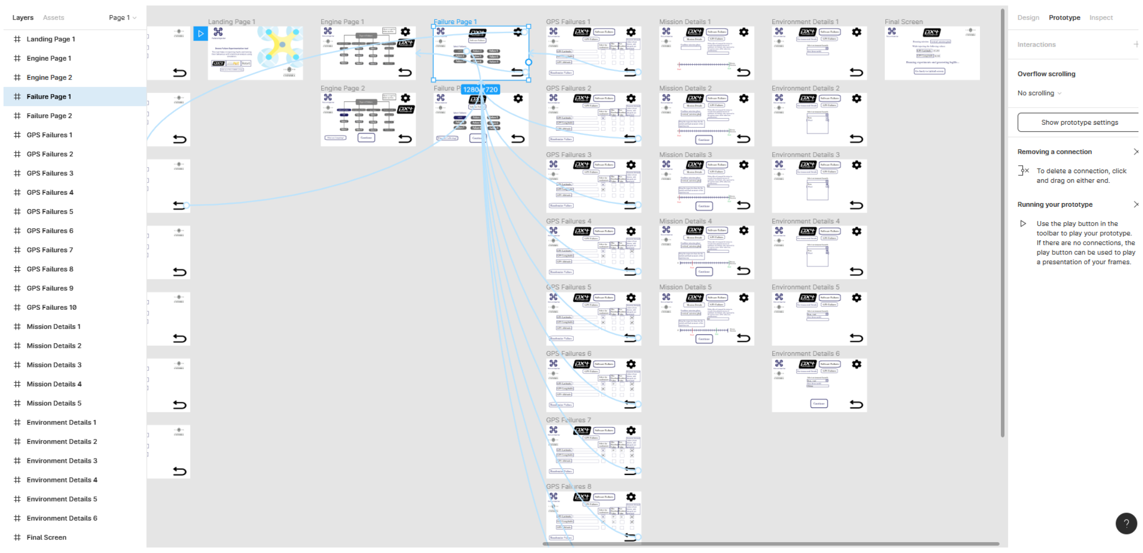
Chapter 6

Tool Implementation

This chapter offers an in-depth explanation of the tool's conceptualization - in the form of an interface prototype - and its implementation. The implementation section approaches both the current stage of the product, the tools used to implement it and what was changed within the PX4 software itself.

6.1 Conceptualization

As mentioned, before starting the implementation of our tool, we began a conceptualization phase that manifested itself as a **Prototype UI**. We accomplished this by making use of a web-based prototyping tool, **Figma**. Figma focuses on providing an accessible way of designing fluid prototypes that can mimic the intended product. Figure 6.1 is an example of how our Figma's work environment looked like throughout the conceptualization phase.



Overflow scrolling

No scrolling

Removing a connection

To delete a connection, click and drag on either end.

Running your prototype

Use the play button in the toolbar to play your prototype. If there are no connections, the play button can be used to play a presentation of your frames.

Figure 6.1: Figma Project Screen

In figure 6.2, we can observe our intended landing page. Here the user can select the flight controller under assessment that he wishes to use and advance to the next page. In the next page the user must choose one of the failures types presented. The available choices are based on the failure types introduced in section 2.2. This can be observed in figure 6.3. Once the user chooses a failure type, the button highlights and an option to advance the injection process appears. Choosing a failure type also unlocks a new button that was intended to be used to showcase the architecture of the system under assessment and more specifically, the module that the selected failure would be targeting. This button and the previously mentioned details can be observed in figure 6.4.

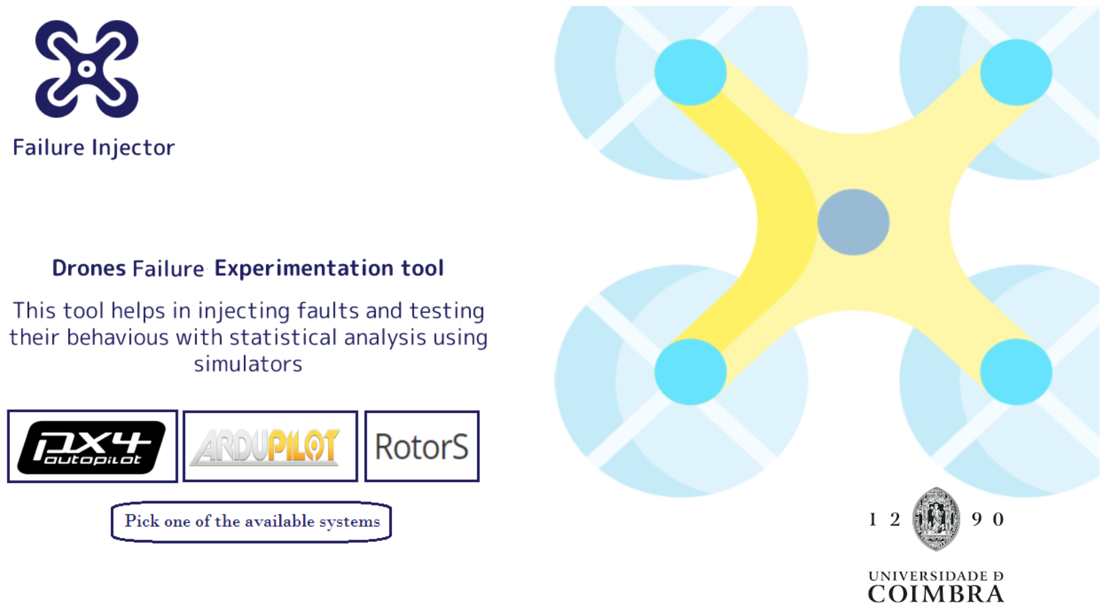

Failure Injector
Drones Failure Experimentation tool
This tool helps in injecting faults and testing their behavious with statistical analysis using simulators
PX4 autopilot
ARDUPILOT
RotorS
Pick one of the available systems
1 2 9 0
UNIVERSIDADE D COIMBRA

Figure 6.2: Prototype Landing Screen

At any time the user can choose to either return to a previous screen to redo a choice, by pressing the arrow icon, or open the options menu, by pressing the gear icon. The options menu was intended to be a simple screen where the user can configure relevant paths and alter the language of the tool. It can be observed in figure 6.5.

Advancing from the failure type choice screen provides the user with the option of choosing a specific failure to simulate. Again, once the user enters the screen every option starts by being greyed out but once the user makes a choice and presses the button corresponding to one of the available options, the option highlights and the user is allowed to proceed. This is reflected in figure 6.6.

The next screen was intended to reflect the choice previously done and would change accordingly. The example provided in figure 6.7 shows that choosing a GPS failure provides the user with options to inject GPS values according to a random range of values or fixed values. The screen was intended to encompass every available injection option for the specified failure. For the presented example, this means that the screen was supposed to handle different types of GPS injections (e.g. GPS freeze value, GPS random value, GPS fixed value). We later decided against this and the final product instead has different screens for each individual injection possible.

The two following screens relate to how the simulated drone and world will behave. The first screen, seen in figure 6.8, also provides a way to define the fault injection interval. In said screen the user can define which mission is intended to be run by the simulated drone, the temporal interval in which the fault injection will occur, and the allowed temporal deviation for the simulated faulty runs. The allowed deviation is based on the reference run, which means that if a simulated faulty run current run-time is superior to the established reference run run-time plus the defined deviation, it is considered that the run failed beyond recovery and it is then aborted. After defining the mission details it is possible to then

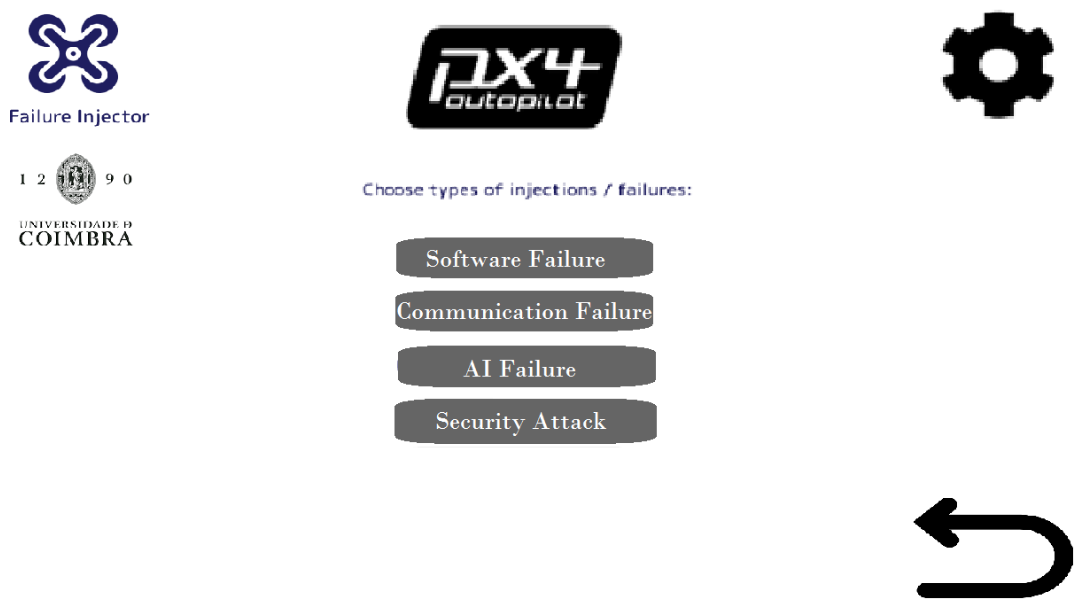


Figure 6.3: Prototype Failures Type Choice Screen (none chosen)

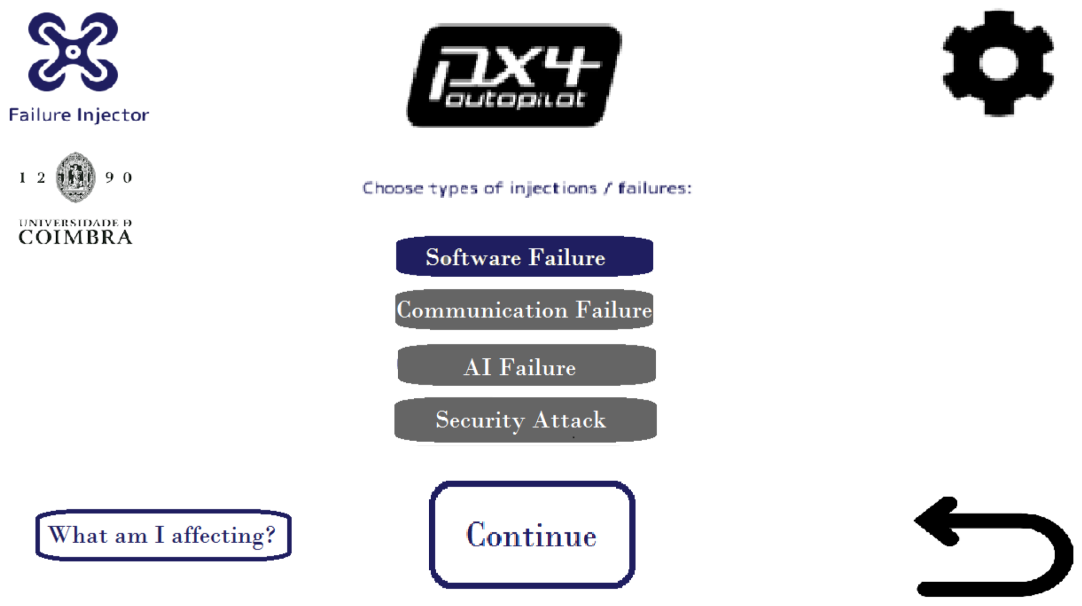


Figure 6.4: Prototype Failures Type Choice Screen (option chosen)

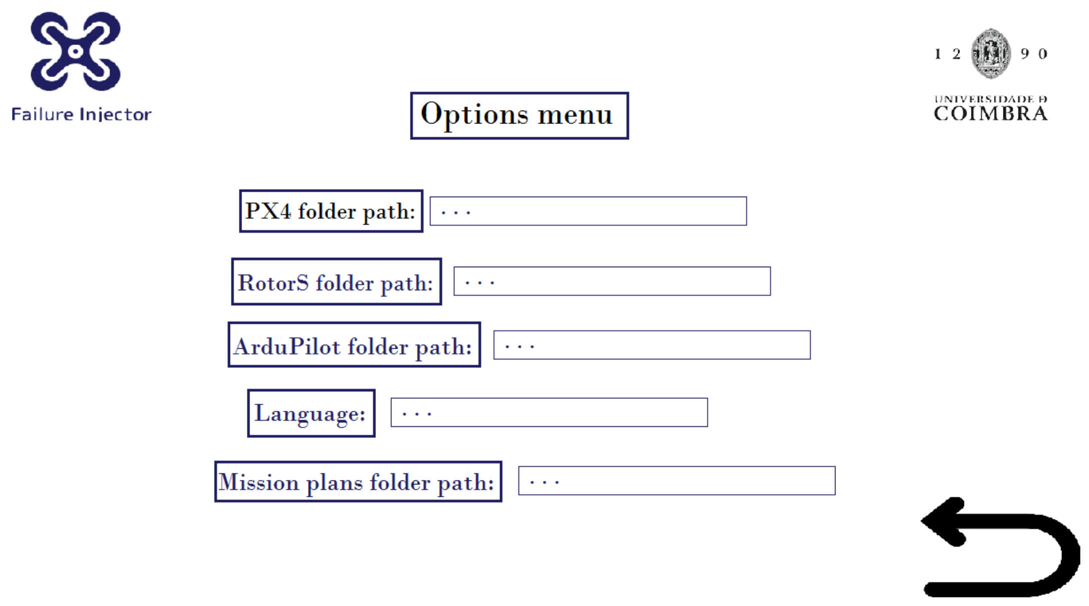


Figure 6.5: Prototype Options Screen

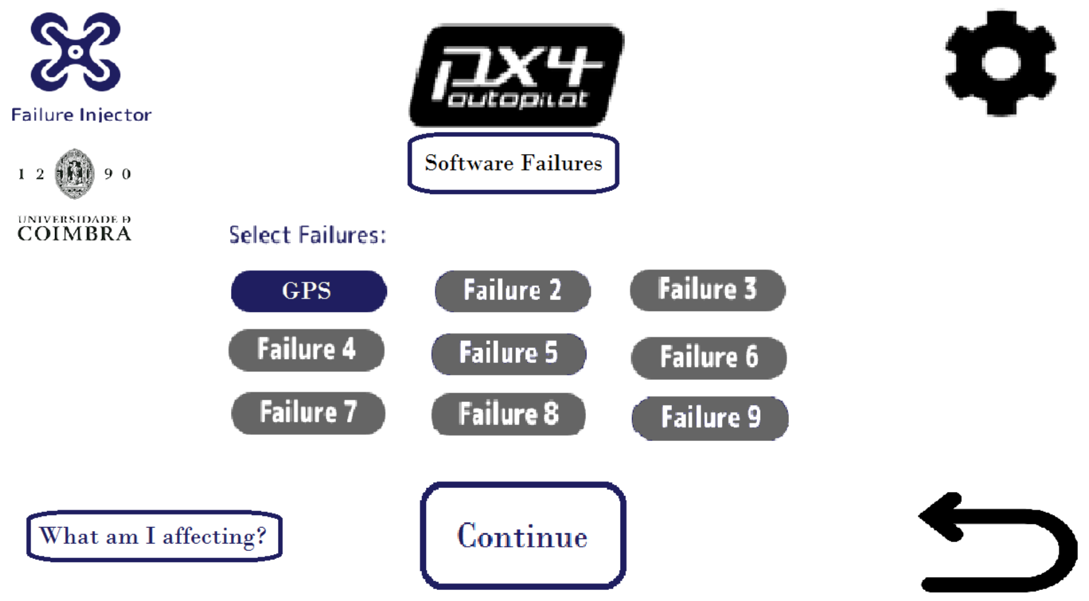
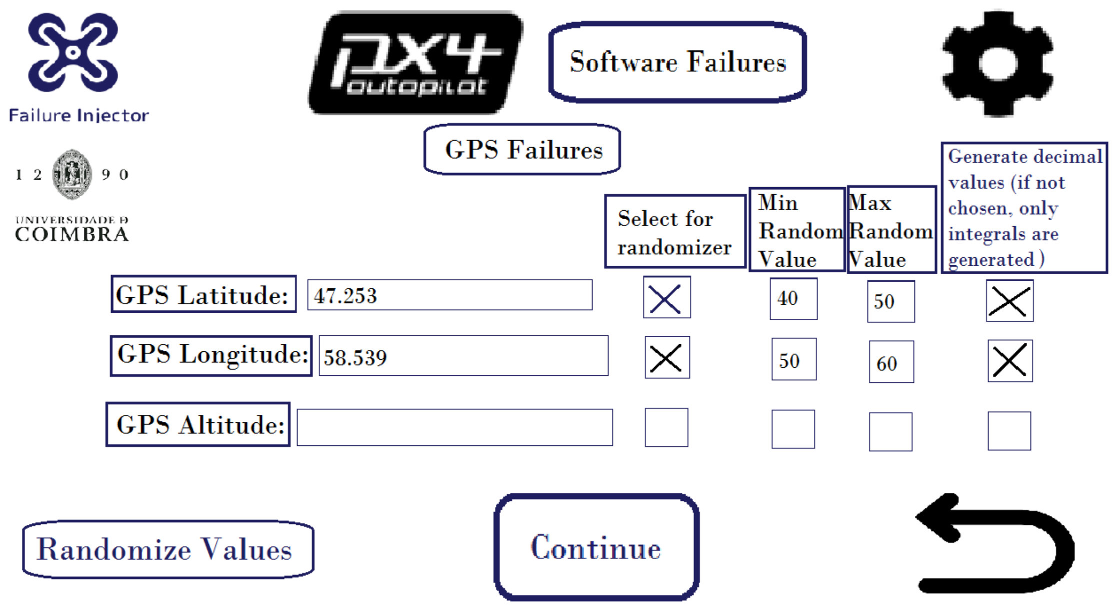


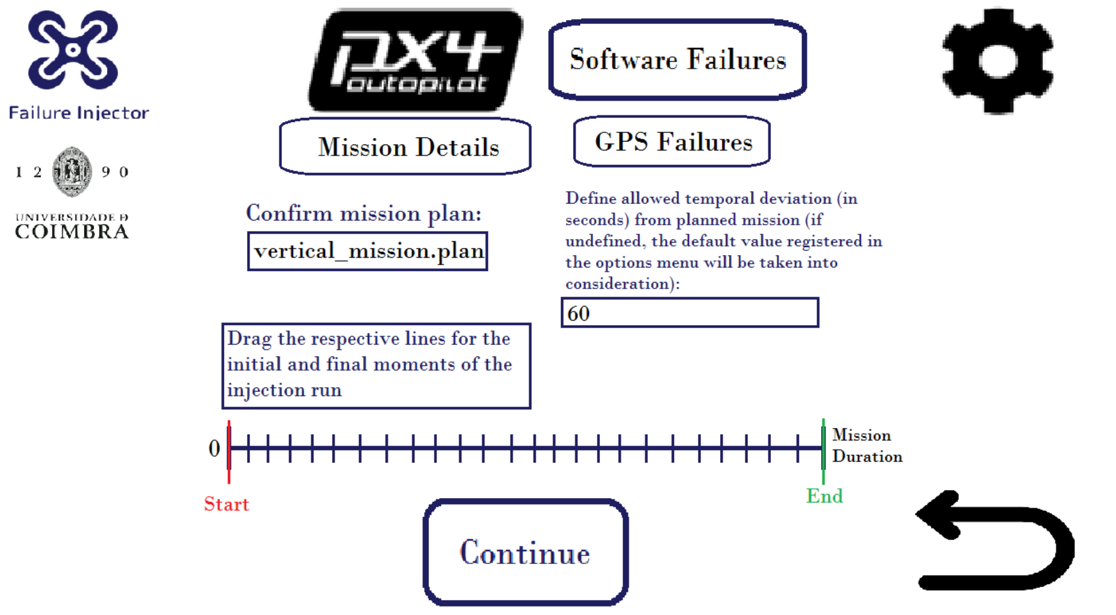
Figure 6.6: Prototype Specific Failure Screen



		Select for randomizer	Min Random Value	Max Random Value	Generate decimal values (if not chosen, only integrals are generated)
GPS Latitude:	47.253	☒	40	50	☒
GPS Longitude:	58.539	☒	50	60	☒
GPS Altitude:		☐			☐

Figure 6.7: Prototype GPS Failure Data Input Screen

define the environmental details as shown in figure 6.9 and figure 6.10. Here the user could select a drone model and some hazards that would affect the simulated world. A big oversight that was later fixed in the developed tool is the lack of an option for selecting a specific gazebo world. The option to select a gazebo world also eliminated the need for environmental hazards in the interface as these could be directly defined in the world's file.



Confirm mission plan: vertical_mission.plan

Define allowed temporal deviation (in seconds) from planned mission (if undefined, the default value registered in the options menu will be taken into consideration): 60

Drag the respective lines for the initial and final moments of the injection run

Figure 6.8: Prototype Mission Details Screen

The last planned screen for a single fault campaign definition can be seen in figure 6.11. At the time we considered this screen valid for its purpose but it was severely redesigned in the final product. For the final product the information that is relevant to the campaign is shown before the experiments start, so that the user can validate the decisions made throughout the campaign definition. After validating and advancing to the following screen,

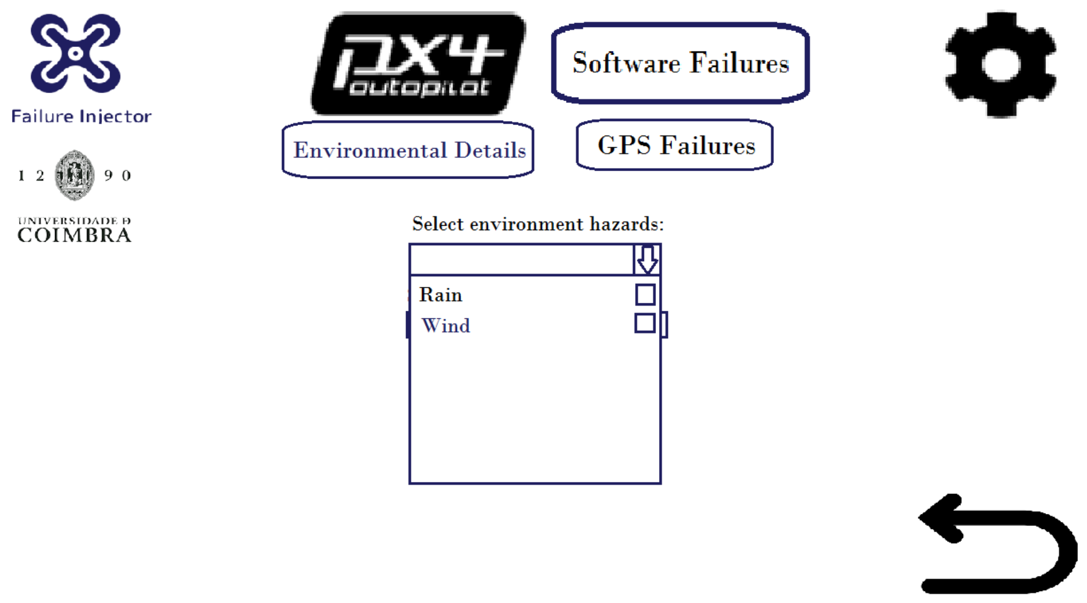


Figure 6.9: Prototype Environmental Details Screen (hazard options)

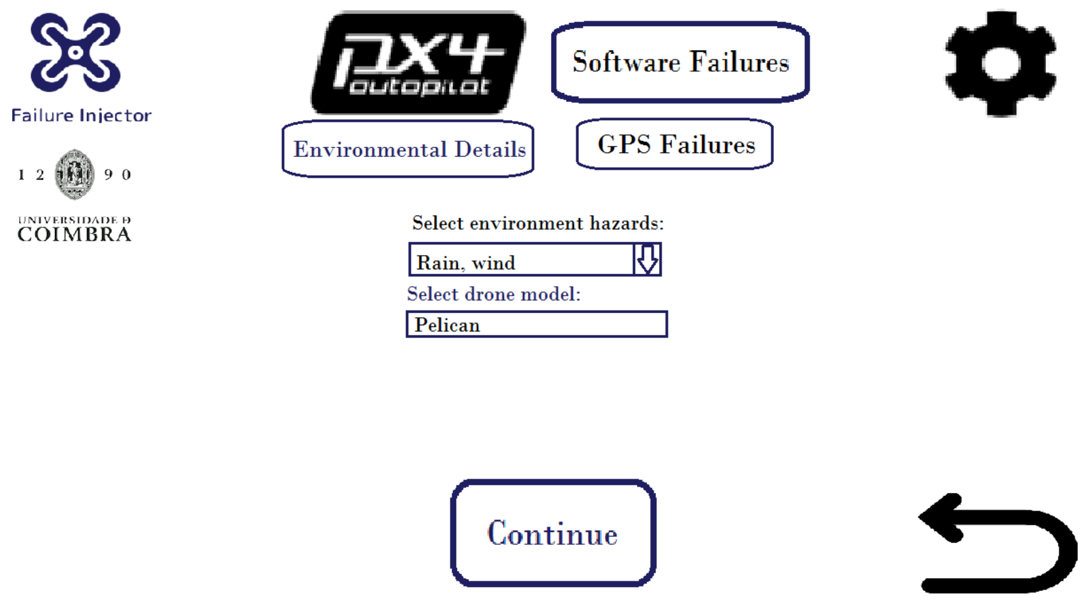
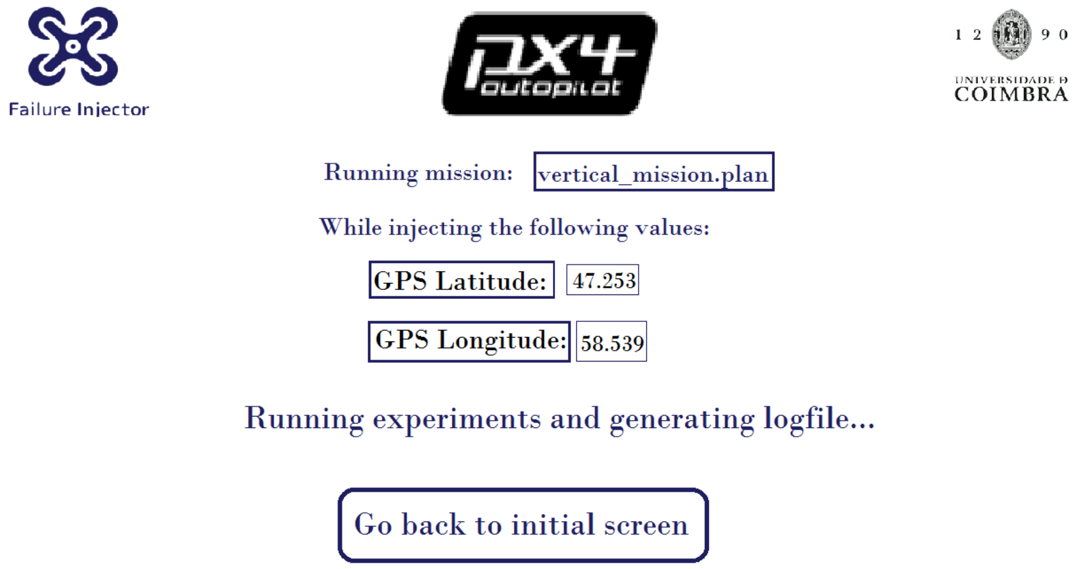


Figure 6.10: Prototype Environment Details Screen

the user is presented with a field that provides pertinent information on the campaign's progress (e.g. which run the campaign is on, each individual run's progress). These changes can be observed in figures 6.20 and 6.21.



Running mission: vertical_mission.plan

While injecting the following values:

GPS Latitude: 47.253

GPS Longitude: 58.539

Running experiments and generating logfile...

Figure 6.11: Prototype Campaign Confirmation Screen

In figure 6.12 a simplistic workflow chart of the interface prototype can be observed.

6.2 Implementation

The current section is divided into a subsection that explains the fault injector's current interface and another subsection that addresses the changes made to the flight controller software and how these changes allow the tool to systematically inject faults.

6.2.1 Tool Interface

The tool was implemented using the integrated development environment (IDE) **Qt Creator**. Qt Creator is known for its cross-platform utility, multi-language support and its ability to simplify GUI application development. These were all appealing aspects to us, as we did not want our injector to be limited by operative systems. An example of Qt Creator's interface can be seen in figure 6.13.

In figure 6.14 we can observe the current landing screen with the options tab open. Here the user can start the definition of a new campaign, or upload a previously saved campaign. A campaign can be saved after the definition process, as shown in figure 6.20, and be reused at a later date. In the upper bar it is possible to observe three different tabs – Options, Tools and Help. The Options tab is used to define relevant paths (e.g. flight controller installation folder or the mission folder), to change the tool language and to change flight controller specific options. Currently, we have two specific settings for PX4 hooked to our tool – being able to change the simulated drone's max speed and the flight controller's GPS Failsafe activation delay - the latter allows the user to bypass the flight controller's GPS error detection for a certain period of time. The Tools tab is used to access additional

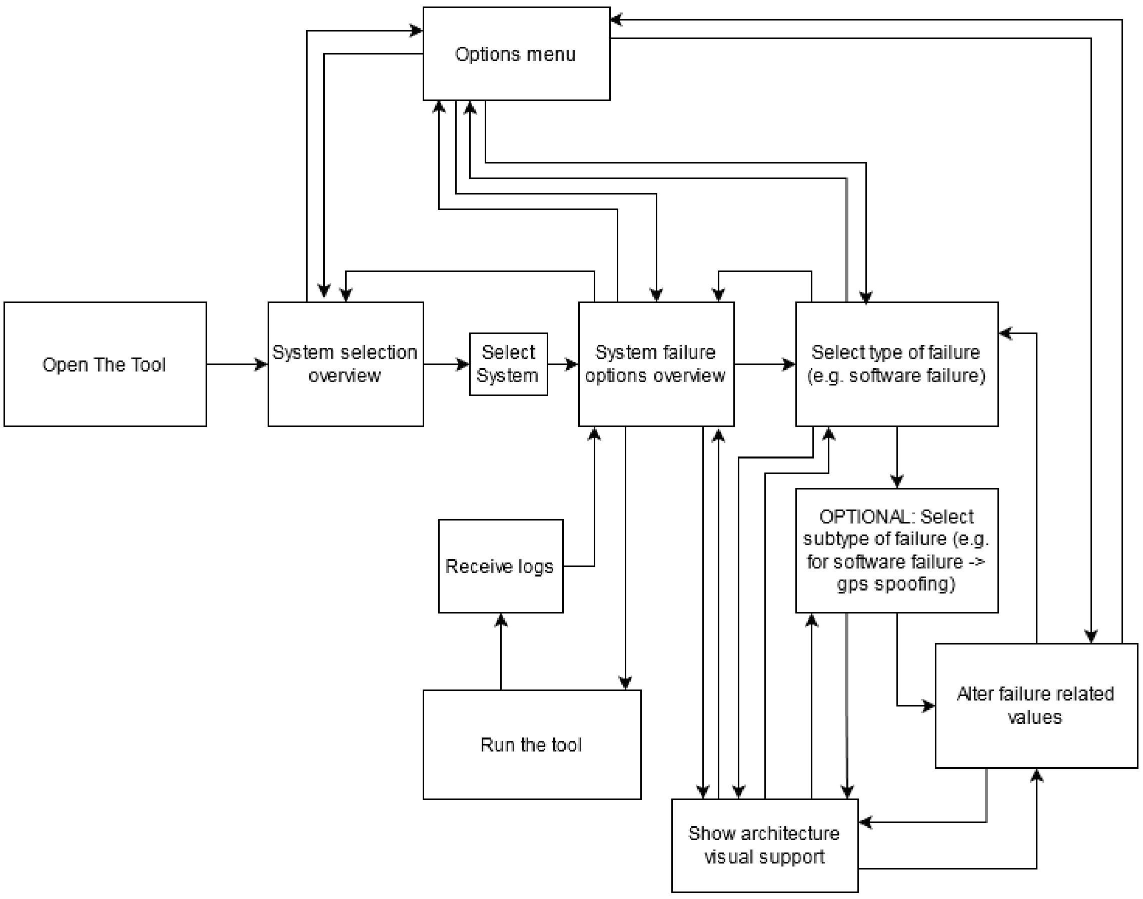
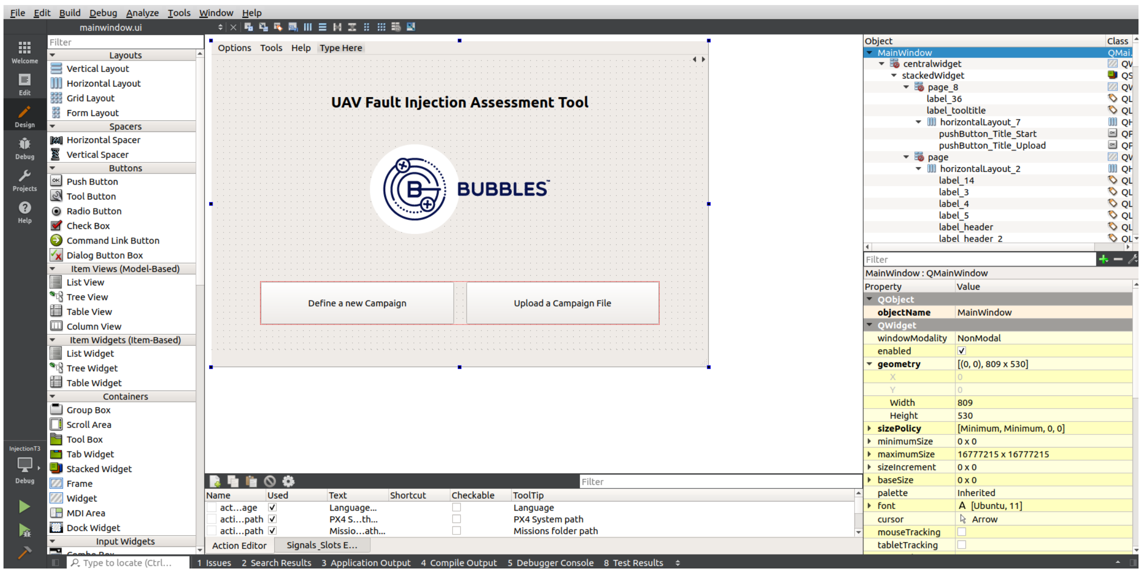

Options menu
Open The Tool
System selection overview
Select System
System failure options overview
Select type of failure (e.g. software failure)
Receive logs
OPTIONAL: Select subtype of failure (e.g. for software failure -> gps spoofing)
Alter failure related values
Run the tool
Show architecture visual support

Figure 6.12: Prototype Workflow



Object	Class
MainWindow	QMai…
centralwidget	QV
stackedWidget	QS
page_8	QV
label_36	QL
label_tooltitle	QL
horizontalLayout_7	QH
pushButton_Title_Start	QP
pushButton_Title_Upload	QP
page	QV
horizontalLayout_2	QH
label_14	QL
label_3	QL
label_4	QL
label_5	QL
label_header	QL
label_header_2	QL

MainWindow : QMainWindow

Property	Value
QObject	
objectName	MainWindow
QWidget	
windowModality	NonModal
enabled	✓
geometry	[(0, 0), 809 x 530]
X	0
Y	0
Width	809
Height	530
sizePolicy	[Minimum, Minimum, 0, 0]
minimumSize	0 x 0
maximumSize	16777215 x 16777215
sizeIncrement	0 x 0
baseSize	0 x 0
palette	Inherited
font	A [Ubuntu, 11]
cursor	Arrow
mouseTracking	☐
tabletTracking	☐

Name	Used	Text	Shortcut	Checkable	ToolTip
act…age	✓	Language…		☐	Language
acti…path	✓	PX4 S…th…		☐	PX4 System path
acti…path	✓	Missio…ath…		☐	Missions folder path

Action Editor Signals_Slots E…

Figure 6.13: QtCreator Project Screen

features (e.g. imbued mission planner or world builder) and the Help tab is used to access information related to the tool. At the moment, the Tools and Help tab functionalities are not implemented, but we already provide tips throughout the campaign definition and simulation to guide the user. Pressing the “Define a new Campaign” button takes the user to the next screen.

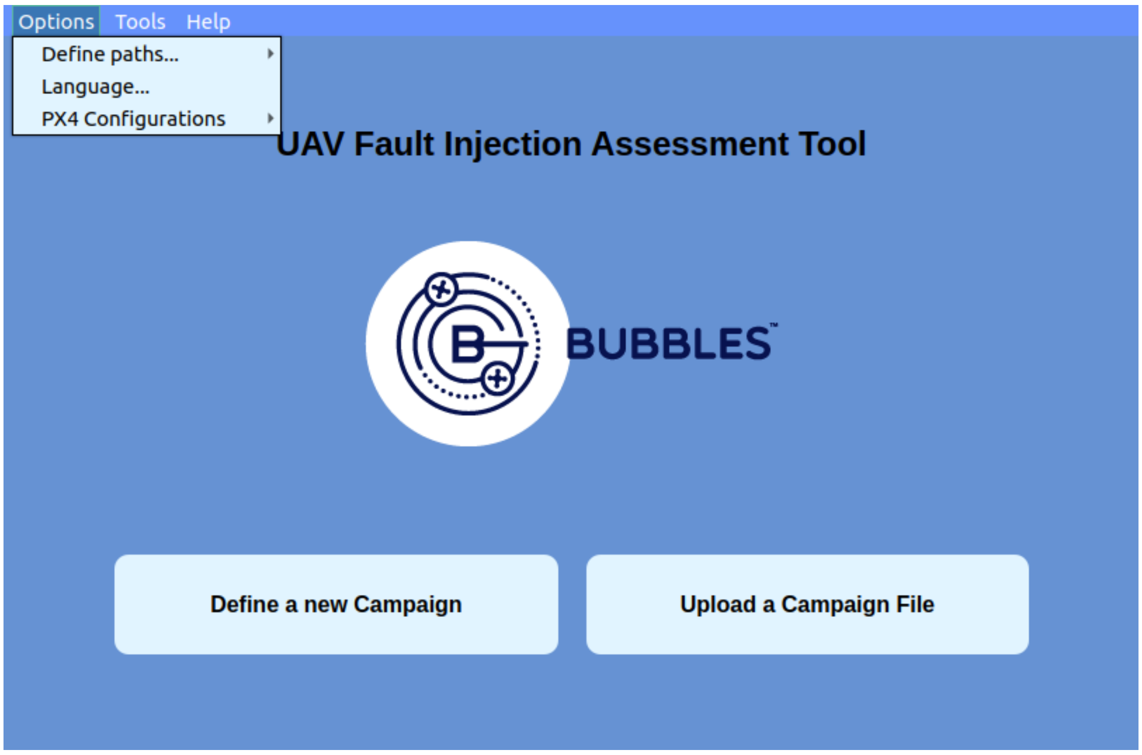


Figure 6.14: Tool Landing Screen

The following screen, shown in figure 6.15, starts the campaign definition process by having the user choose the flight controller under assessment. It is possible to see that although not identical, the screen was based on the prototype interface since it presents some similarities with the screen shown in figure 6.2. As was previously defined, we are currently only working with PX4, but we may expand the available flight controllers in the future and as such, our tool takes that into account. We also present the user with a campaign definition workflow that can be highlighted in order to see the available tip for the current page. Selecting a flight controller advances the user to the next screen.

After defining the flight controller under assessment, the user must choose the drone model that will be simulated, as seen in figure 6.16. The list of available drones is dependent on which drones are supported by the flight controller. Currently, we have three options available – the 3DR Iris+, the 3DR Solo and the Typhoon H480. By default, the tool defines the 3DR Iris+ as the drone under assessment, but this can be changed through the drop-down list. After selecting the desired drone, the user can advance to the following screen by pressing the “Next” button. Although not shown, every button highlights when hovered on and provides a tip on what its function is.

In the Mission and Environmental Details screen, seen in figure 6.17 the user can choose a mission plan, select a gazebo world and define the allowed deviation from the reference run. As the option to plan missions in the tool is not currently implemented, the missions need to be written externally. The missions follow the **Mavlink Plain-Text File Format** that can be found at [44]. As previously addressed, we intended to use QGroundControl

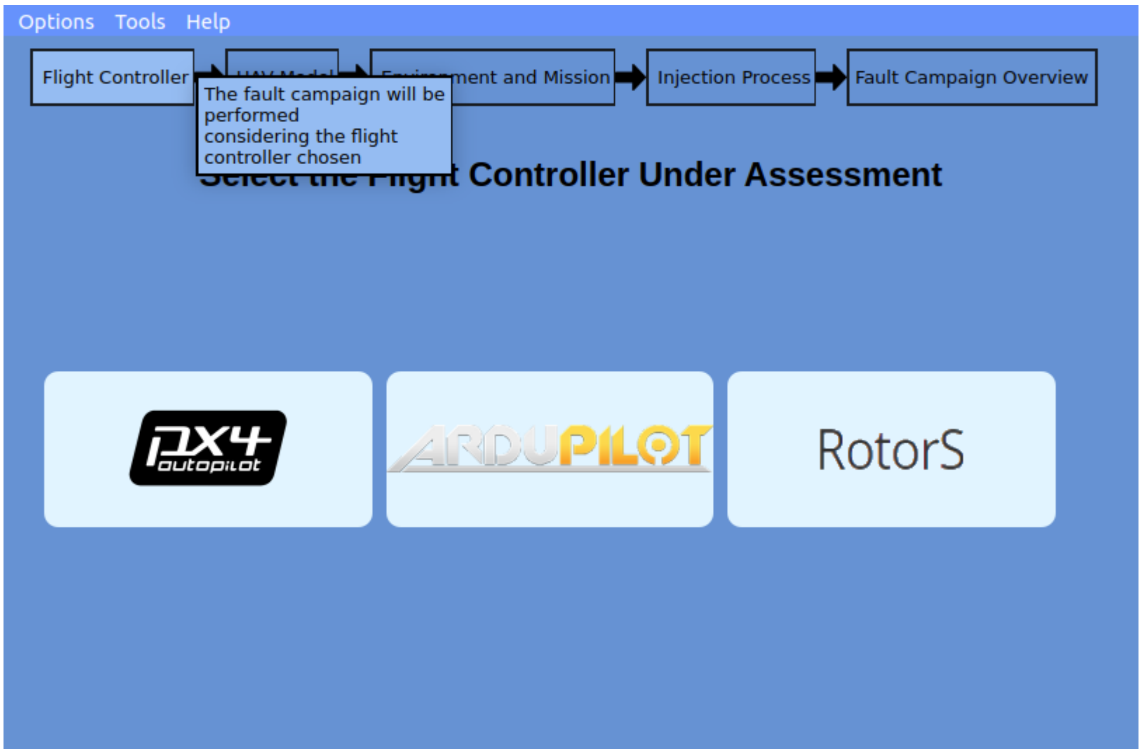


Figure 6.15: Tool Flight Controller Definition Screen

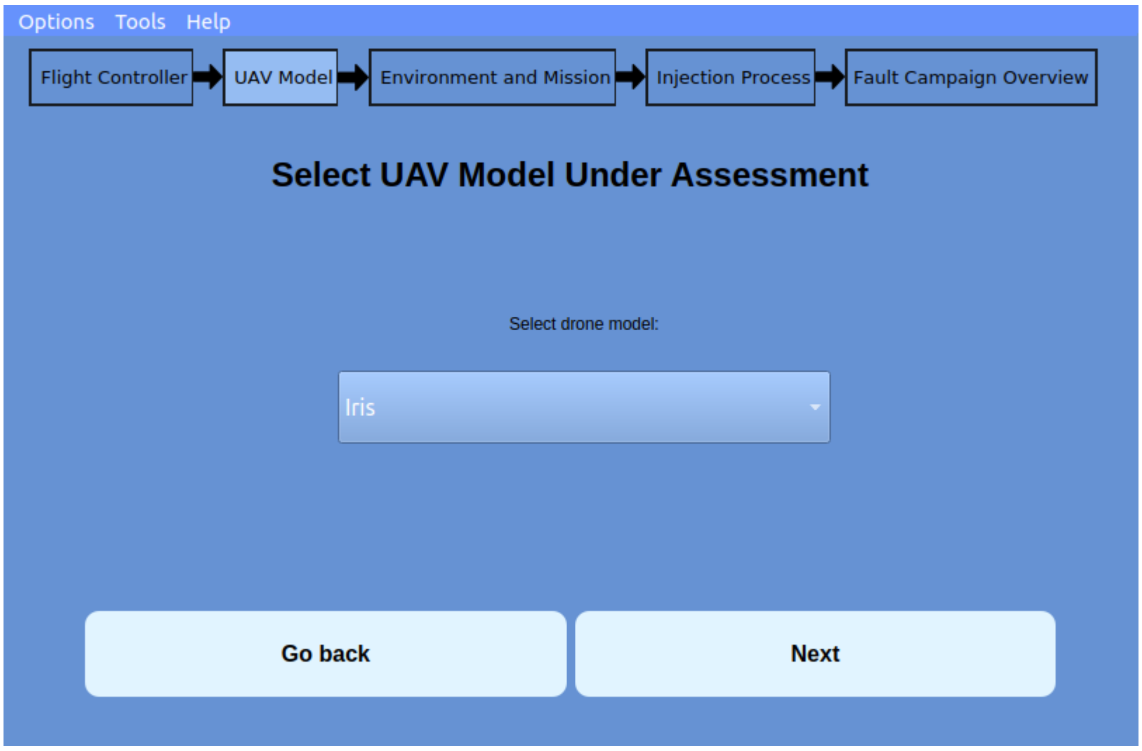
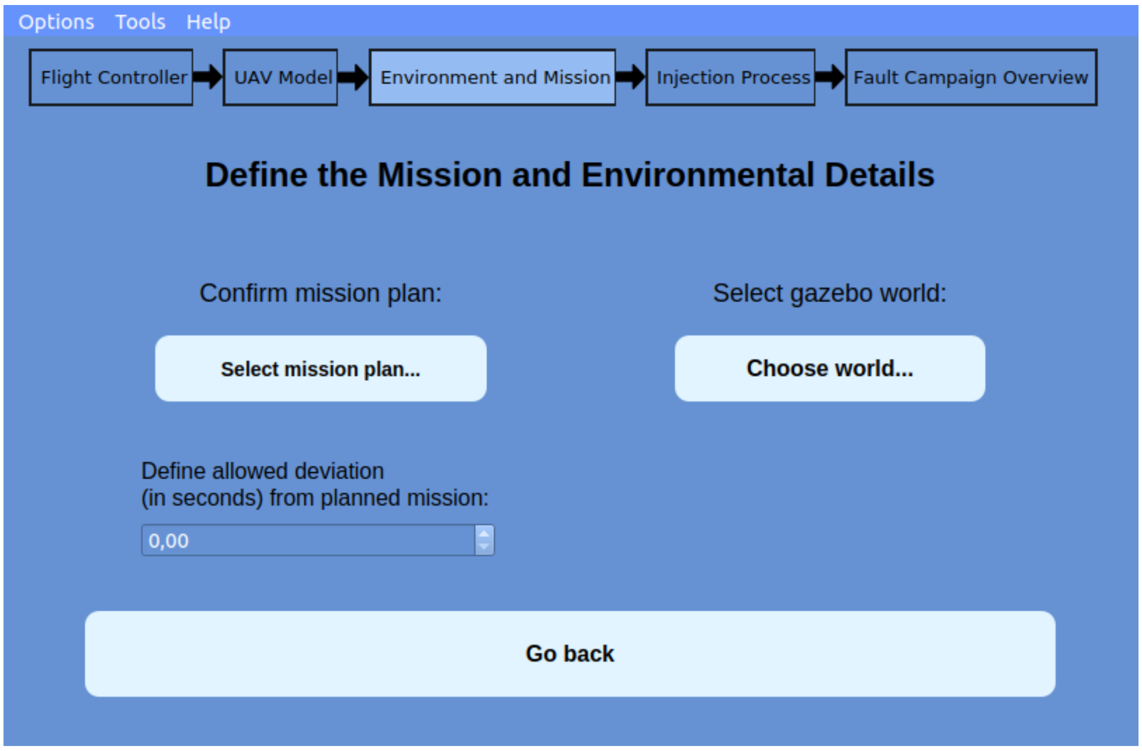


Figure 6.16: Tool UAV Model Definition Screen

to provide support (e.g. generate and upload missions) but found its API to be severely lackluster and as such, switched to the simpler DroneKit, which is compatible with vehicles using the MAVLink protocol and presents a much more accessible API. Although missions need to be written externally, since our intention is to provide support to the BUBBLES project we wrote a script that converts BUBBLES GPS coordinates JSON files into valid mission files. The Gazebo worlds must also be written externally, but a wide variety of examples are already provided with the flight controller. Gazebo also has a wide array of tutorials available on its website [45], as well as an active community that constantly creates content to provide help to both beginners and advanced users. As previously explained during section 6.1, the defined deviation allows users to define a time limit that injection runs must oblige. Filling every field allows the "Next" button to appear and the user can then proceed with the campaign definition process.



Define the Mission and Environmental Details

Confirm mission plan:

Select gazebo world:

Define allowed deviation
(in seconds) from planned mission:

0,00

Figure 6.17: Tool Mission and Environmental Details Screen

In the following screen, shown in figure 6.18, the user can configure the injection campaign. Here it is possible to define the number of runs the campaign will have, define which faults will be injected and verify which faults are being injected. The faults that can be injected are divided into three categories. The fault type has the highest hierarchy and is based on the failures types introduced in section 2.2. Each type has a set of targets, with the one highlighted in figure 6.18 being a GPS failure. Finally, for each fault type and fault target combination there are a number of fault sub-types. Currently, only faults that target the GPS module are implemented. Faults that target the GPS can be either interpreted as a software failure or a security attack, depending on the context. In order to reflect this we have scenarios for both available in the tool's interface, but the implementation behind them is the same.

The available faults are classified as follows:

- **Software Failures**:
  - **Fixed values**: The GPS values are maintained at a fixed value defined by the

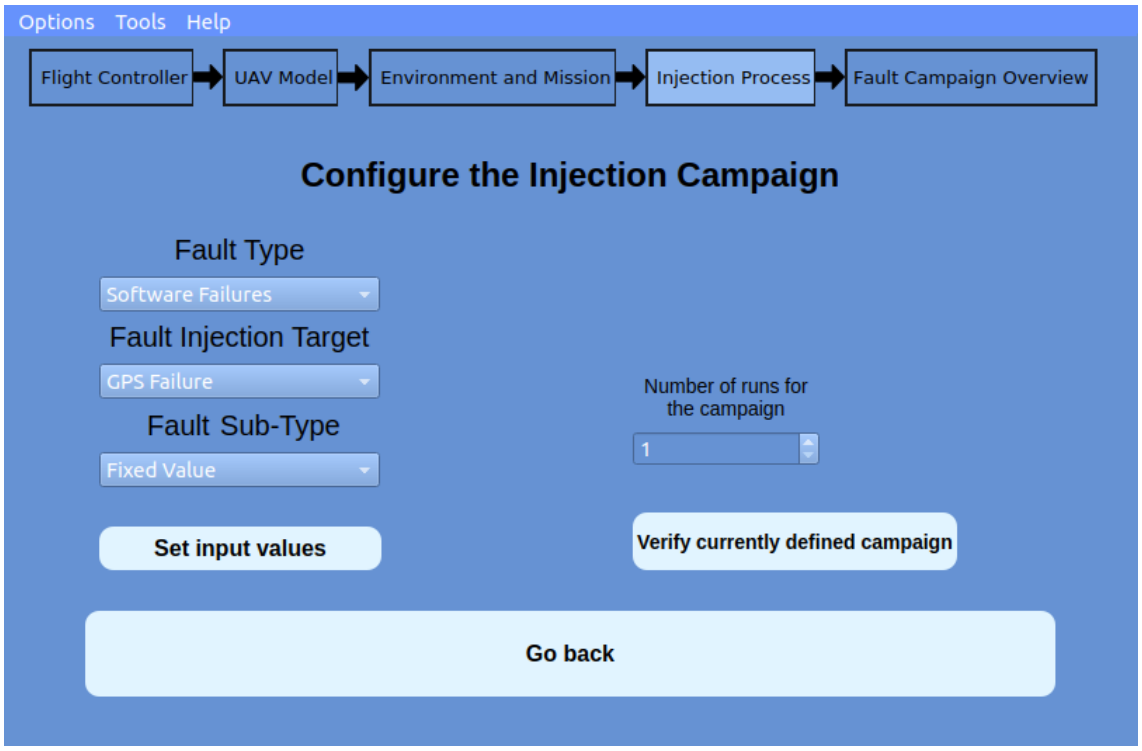
user.

  - **Delayed values**: The GPS values are delayed using a time delay specified by the user.
  - **Freeze values**: The GPS values are maintained at a fixed value defined by the values present in the GPS when the fault injection started.
  - **Random value inside a range**: The GPS values are set to random values inside a range defined by the user.
  - **Min/Max value**: The GPS values are set to the maximum or minimum value of the variables that hold them depending on the user's choice.

- **Security Attacks**:
  - **Random Longitude**: Tampers the longitude reading with a random value in a valid range of values (-180 to 180).
  - **Random Latitude**: Tampers the latitude reading with a random value of a valid range of values (-90 to 90).
  - **Random Position**: Tampers the position readings with random valid values on the three axes (latitude, longitude and altitude).
  - **GPS delay**: Does not tamper the GPS values, but delivers them with a certain delay (defined by the user).
  - **Force UAV landing**: Tampers the altitude values with (slightly) higher values than the real one, trying to force an unplanned landing.
  - **Hijack with a second UAV**: Tampers the position readings with values from another drone trajectory, on the three axes (latitude, longitude and altitude)
  - **Hijack with attacker's specified position**: Tampers the position readings with values from static position given by the attacker, on the three axes (latitude, longitude and altitude).

For each combination of type, target and sub-type there is a different screen for the user to input the relevant data. This screen is accessed through the "Set input values" button and is inaccessible whenever one of the three fields is empty. These screens are simplistic but we found that it was better to have one screen for each possible combination rather than to have one screen for each possible target as was mentioned in the previous section. A possible input screen can be seen in figure 6.19. In this case, it is the screen used for a GPS Software Failure using an interval of random values. After defining at least one fault a button that allows the user to advance appears. Advancing to the next screen marks the end of fault campaign definition and the user can now see an overview of the relevant decisions done throughout the definition of the campaign as seen in figure 6.20.

In this screen, it is possible to return to the first screen, start the injection campaign or save the campaign in a file for future use. To use the file in subsequent sessions, the user only needs to upload the file using the button in the landing screen, as previously stated. Initiating the campaign starts the simulator with the respective chosen drone and world. The reference run is always ran first, with the following runs corresponding to the number of runs defined in the campaign. Besides the simulator, the tool also remains open providing helpful information related to the campaign's progress - this can be seen in figure 6.20. During the campaign, it is possible to abort it, cancelling all future runs that have not yet been ran. It is also possible to pause the campaign in between runs – in the case the user wants to pay closer attention to some information that is being displayed - using the "Halt run" button. After the campaign has ended or has been forcibly terminated, new buttons



Configure the Injection Campaign

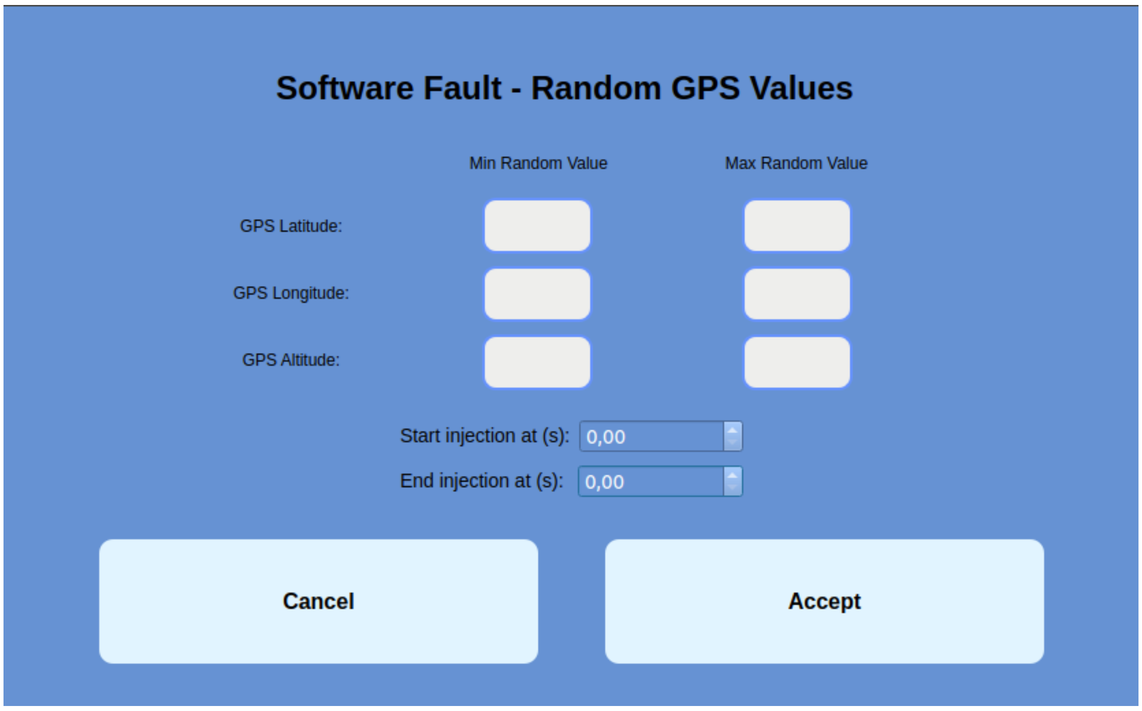
Fault Type: Software Failures

Fault Injection Target: GPS Failure

Fault Sub-Type: Fixed Value

Number of runs for the campaign: 1

Figure 6.18: Failures Definition Screen



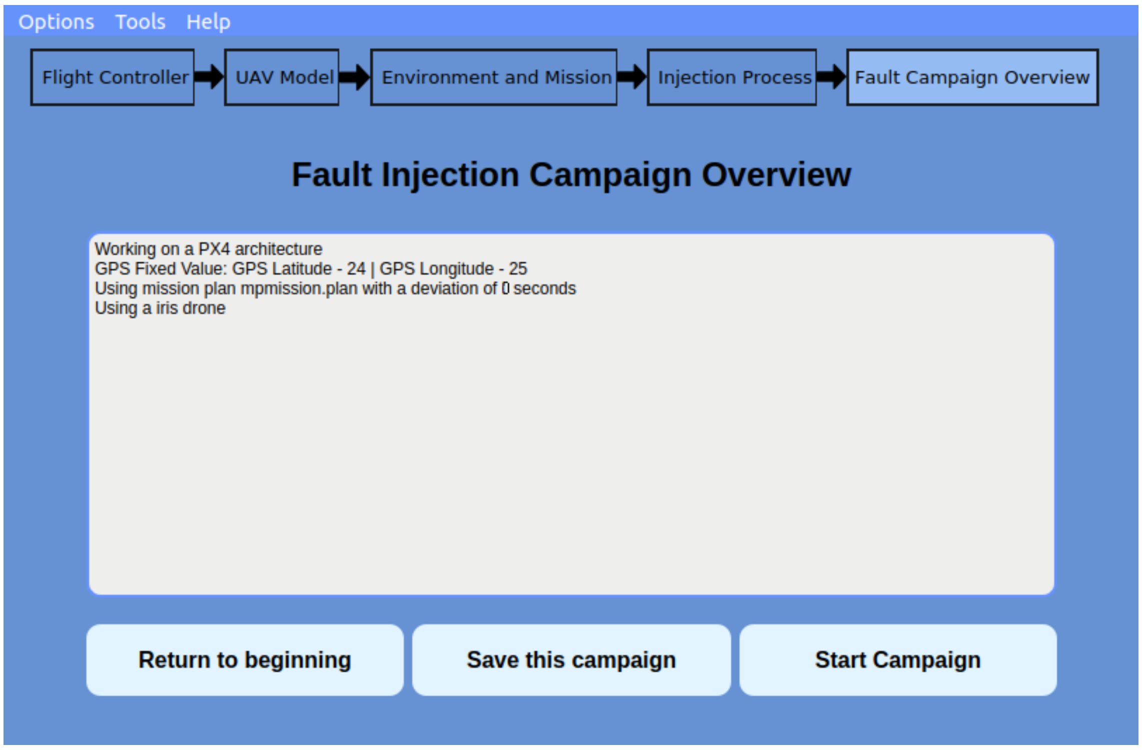
Software Fault - Random GPS Values

	Min Random Value	Max Random Value
GPS Latitude:		
GPS Longitude:		
GPS Altitude:		

Start injection at (s): 0,00

End injection at (s): 0,00

Figure 6.19: Random GPS Values Injection Screen



```
Working on a PX4 architecture
GPS Fixed Value: GPS Latitude - 24 | GPS Longitude - 25
Using mission plan mpmission.plan with a deviation of 0 seconds
Using a iris drone
```

Figure 6.20: Campaign Overview Screen

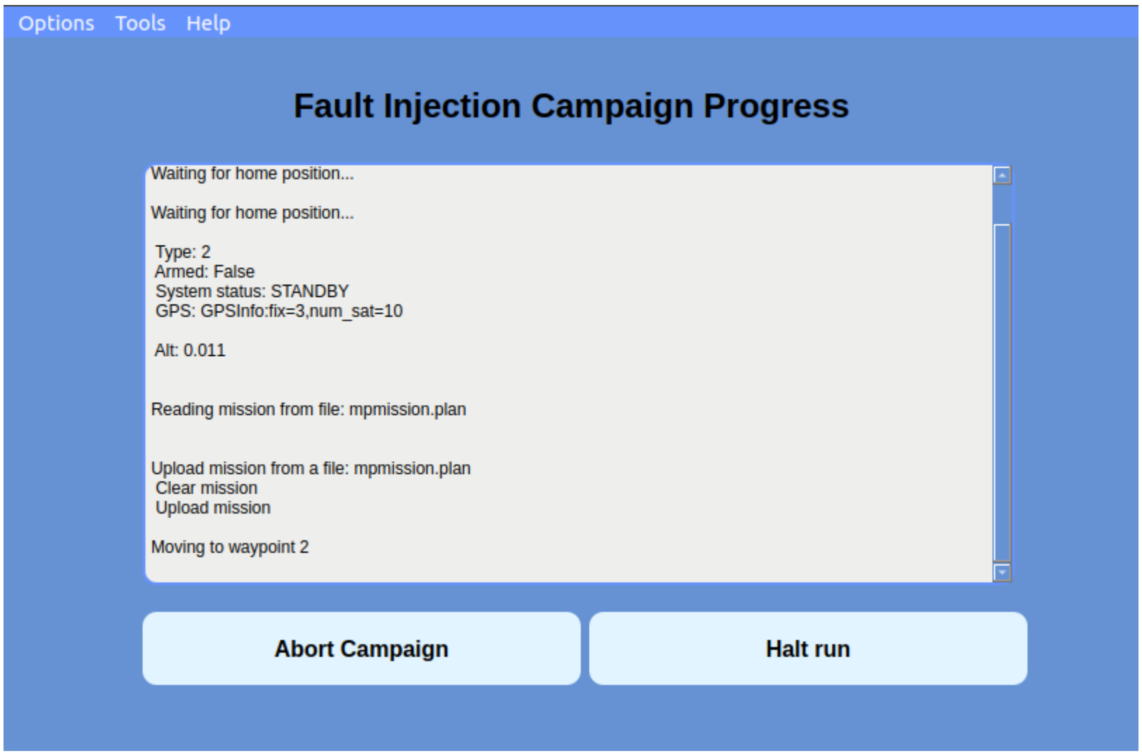
appear which allow the user to return to the first screen, see some simple graphs about the campaign or save the log files of the faulty runs, as seen in figure 6.22. The reference run's log file is always saved so that the user can analyze it later. The log files are generated by PX4 and must be analyzed using an external tool provided by the company [46]. The result graphs can be seen by pressing the "See result graphs" button which takes the user to the screen shown in figure 6.23. The screen only shows the trajectory of the reference run and the trajectory of a random faulty run. They are a very superficial way of resuming the campaign as they don't take anything into consideration besides the trajectory. This means that even if the drone received wrong values; it is possible that they don't affect its trajectory as the case presented shows.

6.2.2 Fault Injections Implementation

To accomplish the fault injection process, our tool invokes the simulated elements (e.g. world and drone), as well as the mission to be ran, as **daemons** (background processes). This allows the environment elements to be centralized in the tool, without the need to directly open additional programs. To convert the user's input into values that could affect the flight controller, we made some changes to its software.

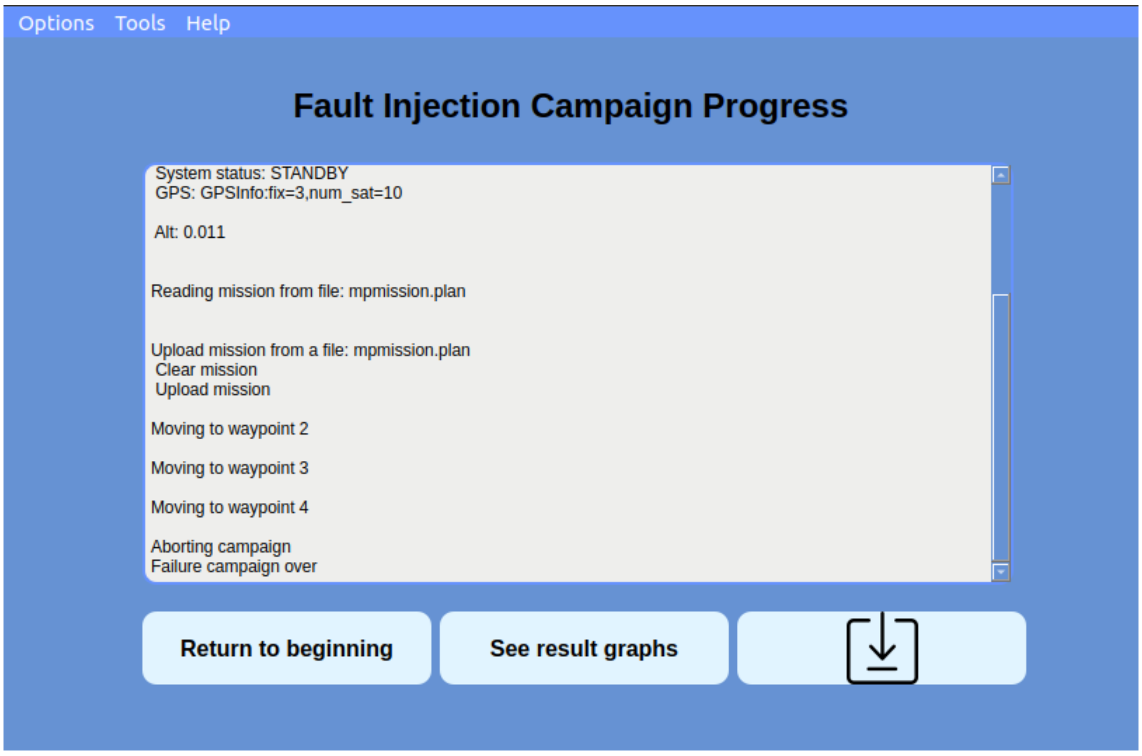
To maximize compatibility we sought to do the least intrusive alterations possible. Our solution was to convert the user's input into a configuration file, that is then read by a newly added function. The data read by this function is then saved, and used in the publishing function present in the sensors module. The publishing function had to be changed so that the GPS values could be replaced by the values present in the configuration file whenever the injection interval was valid.

Besides the changes mentioned, we also added a new type of message to the software that allows the sensors module to know when the drone has started to fly. This is important



```
Waiting for home position...

Waiting for home position...

 Type: 2
 Armed: False
 System status: STANDBY
 GPS: GPSInfo:fix=3,num_sat=10

 Alt: 0.011


Reading mission from file: mpmission.plan


Upload mission from a file: mpmission.plan
 Clear mission
 Upload mission

Moving to waypoint 2
```

Figure 6.21: Campaign Progress Screen (campaign running)



```
 System status: STANDBY
 GPS: GPSInfo:fix=3,num_sat=10

 Alt: 0.011


Reading mission from file: mpmission.plan


Upload mission from a file: mpmission.plan
 Clear mission
 Upload mission

Moving to waypoint 2

Moving to waypoint 3

Moving to waypoint 4

Aborting campaign
Failure campaign over
```

Figure 6.22: Campaign Progress Screen (campaign aborted)

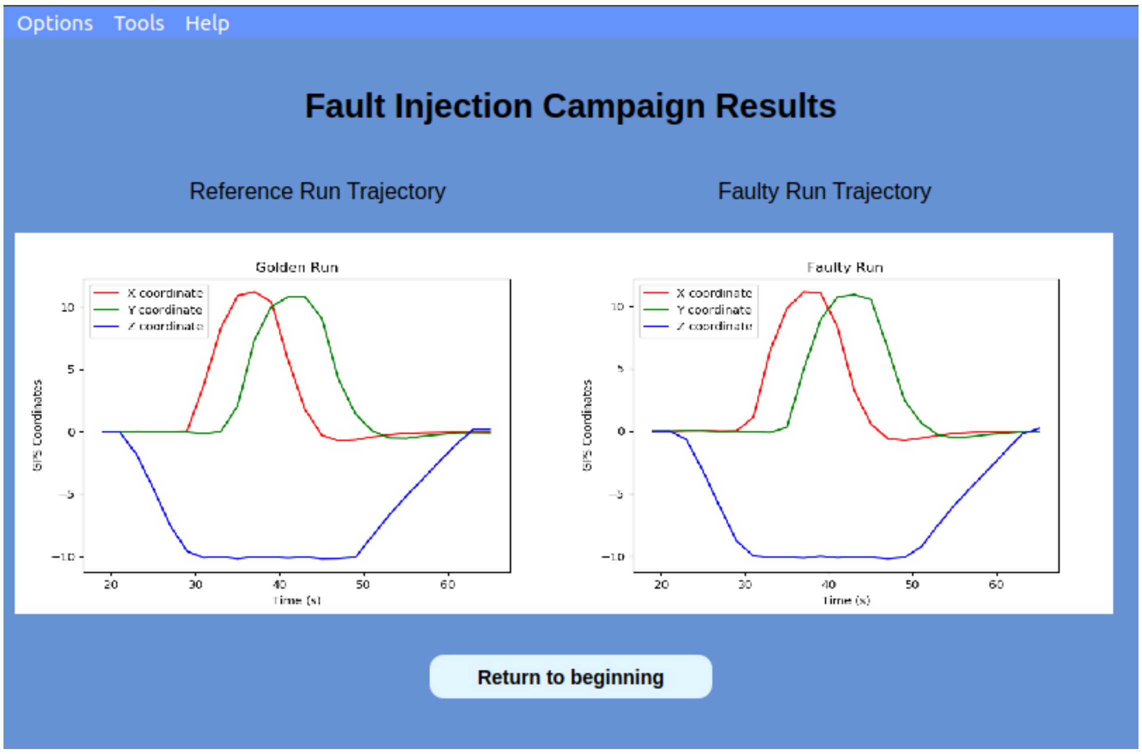


Figure 6.23: Campaign Results Screen

since the injection window is relative to the moment the drone takes off and not to the moment the simulation starts.

Chapter 7

Experimental Setup

This chapter provides an insight on how the experiments were used to validate the implemented GPS failures. The experimental setup is strongly based on the experiments performed in [36]. The experiments were done using the implemented tool and the generated logs were then analyzed by PX4's log analyzer [46].

7.1 Flight Mission

As previously mentioned, the flight mission is closely based on the mission executed in [36], with the main difference being the height of flight - ours being 5 meters instead of 1.5 meters. The mission has a total distance of 130 meters and can be split into three main stages:

- **Takeoff**: From the drone's initial position, it moves vertically to the first waypoint, at 5m of altitude.
- **Horizontal Line**: After takeoff, always at the same height, the drone flies 30m forward in a straight line, and then goes back to the first waypoint's position. This is done two times.
- **Landing**: When the Horizontal Line's second lap ends, the drone lands on the same position from where it took off.

The mission takes around 60 seconds to be completed, at a maximum speed of 5 meters per second, from the moment the drone is armed to the moment it is disarmed. This speed is configurable through the tool, but 5 m/s is PX4's default value. No hard value was set for the allowed mission deviation, meaning the missions could potentially run indefinitely. In one occasion the mission was aborted manually as it went over 15 min with no sign of recovery.

Figure 7.1 gives an overview of the drone's trajectory during the reference run. This image was generated by the previously mentioned log analyzer tool [46].

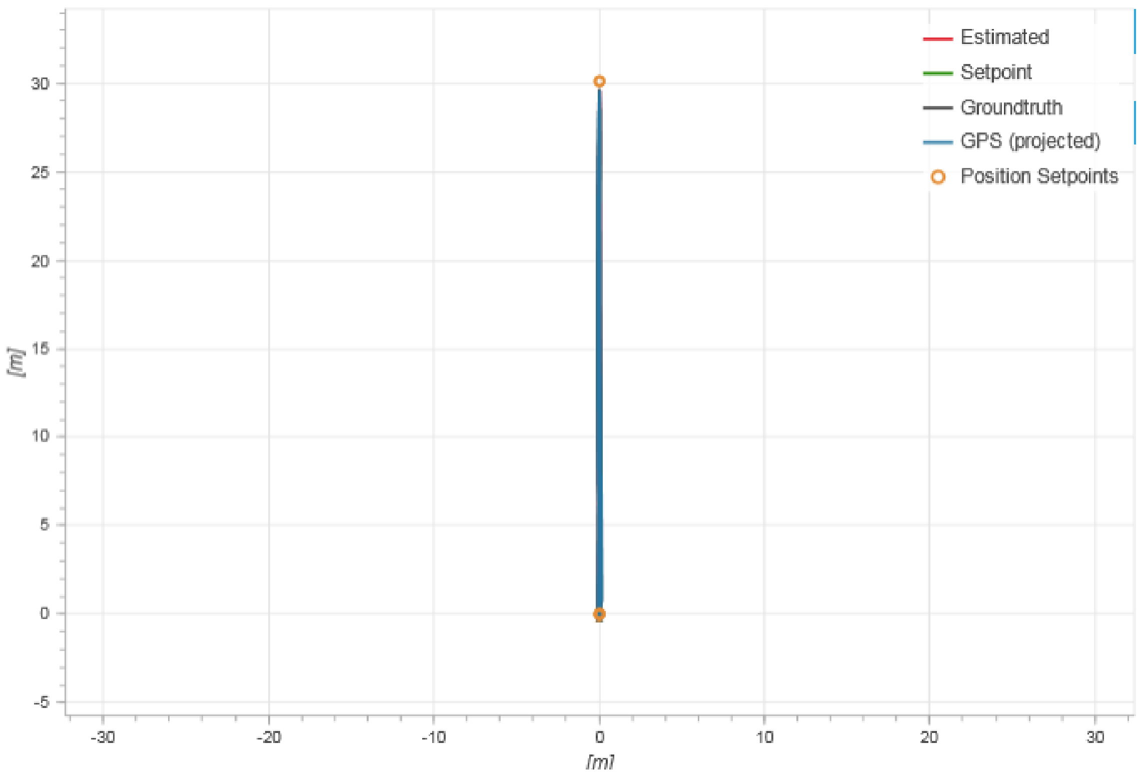


Figure 7.1: Reference Run X and Y axis trajectory

7.2 Testing Model

To replicate the effect of a GPS Failure on the SUA, the implemented faults are present in the sensors hub module. The GPS data is calculated by the GPS driver and then sent to the sensors hub, where the values are tampered with. Figure 5.1, although being the latest published architecture image, does not show the mentioned connection properly as there is no arrow pointing from the GPS driver to the sensor hub. The official GitHub [47] however, properly defines this connection.

Our testing model is defined by three key aspects: **trigger**, **duration** and **type**. The trigger is timed based and is the same throughout all test runs. In every run, the fault is injected 20 seconds after the drone has taken off. The injection duration varies between 5, 15 and 45 seconds. The type is divided into three categories:

- **Freeze Value**, position values read when the injection starts are given during the injection's duration.
- **Fixed Value**, position values are equal to a valid and fixed GPS location.
- **Delay Value**, position values stay the same but are sent with a delay.

We believe that these 3 fault types best summarize the implemented faults.

7.3 Experimenting Process

The experimenting process was done as follows: a fault injection file is prepared using our tool and together with the previously written mission file, both are uploaded to the drone. Each different combination of fault injection duration and fault injection type can be done in one campaign as the tool allows a number of runs to be set. After each campaign ended, the logs were saved locally and later analyzed. This was done for each possible combination.

In order to get valid results, PX4's GPS Failsafe activation delay had to be set to the maximum value - 100 seconds - or else the flights would be aborted. The existence of a failsafe shows that the flight controller is prepared to handle unexpected values but our scenarios assume that the failsafe has the highest possible delay value, either by human manipulation or software failure. However, we briefly provide an insight on what the expected values are when the flights are under the influence of the lowest failsafe activation value.

Before starting the injection campaigns, a few reference runs were performed as to obtain reference values. As it is next to impossible to exactly simulate a real scenario due to external interference (e.g. computer processing power), the initial references runs were used to define acceptable intervals of deviation. The theoretical values for the defined mission are, as mentioned, a distance of 130 meters and duration of 60 seconds. The acceptable deviations gathered from the reference runs were 3.4 meters and 7.7 seconds, respectively. As a distance deviation is more impactful than a duration deviation, only the distance deviation is used to define a valid distance margin. However, we still consider both distance and duration deviation valid ways to deduce that a failure has occurred, as some fault types have a bigger impact on flight duration rather than flight distance. As such, arbitrarily we have defined that for duration we will use the total duration of the mission times 1.5 to define margins. This means that the values that define the normal margin are a deviation 3.4 meters for distance and a duration lower than 90 seconds. In essence, any run whose values deviate from the theoretical distance values by less than 3.4 meters or have a duration lower than 90 seconds are considered unaffected by the fault.

Whenever a run's values, for either duration or distance, go over the defined normal margin, it is considered that a failure has occurred. To make a better distinction between minor and major failures, we have also defined a safe margin. The safe margin was arbitrarily defined as being two times the normal margin - a deviation of 6.8 meters for distance or a duration higher than 180 seconds. If the run's deviation values go over the safe margin, it is considered that a major failure has occurred. However, if they go over the normal margin but not the safe margin, it is considered that a minor failure has occurred. This classification is summarized in table 7.1.

Table 7.1: Results Classification

Classification	**Distance traveled (m)**	**Run Duration (s)**	**Description**
Normal	126.6 <= distance <=133.4	0 <= duration <= 90	deviation <= normal margin
Minor Failure	123.2 <= distance <126.6 or 133.4 <distance <= 136.8	90 <duration <= 180	normal margin <deviation <= safe margin
Major Failure	distance <123.2 or distance >136.8	duration >180	deviation >safe margin

Chapter 8

Result Analysis

In this chapter, the gathered results, documented in appendix A, are analyzed and used to validate the implemented GPS faults. The values gathered in the appendix were taken from the log files generated by PX4. In some occasions, the injected faults compromise these log files and as such, the data might not reflect the real run. To mitigate this, we timed each run with an additional external timer but could not gather distance values in an external manner. For each combination of fault injection duration and type, we ran 10 experiments. The analysis approaches both the injection duration impact and the injection type impact. A brief section is also presented to provide some insight into PX4's GPS Failsafe mechanism.

8.1 Runs Comparison

To test the SUA under three distinct situations, we enacted GPS Failures with three different fault values - **Freeze**, **Delay** and **Fixed**. Although similar, these three types can cover a wide array of scenarios (e.g. Hijack with attacker's specified position, Force UAV Landing).

The Freeze type doesn't allow for much variation, in this case, the GPS values are maintained at a fixed value defined by the values present in the GPS when the fault injection starts. For the Delay type, we chose a delay of 5 seconds. This means that every GPS value is delivered with a 5 second delay for the duration of the fault injection. Since the timer for the fault injections is internal, this fault injection ends up lasting longer than expected, as the 5 second delay is not taken into consideration by the internal clock. Lastly, for the Fixed type, we inject a fixed coordinate with a latitude value equal to 44 and a longitude value also equal to 44. This type of experiment could be replaced by a fault type that provides random values, but we believe that the main aspect we are testing is how the system handles a sudden change in value.

The **Freeze** type injection had the most lackluster results. This is expected, as this type of injection provides the drone with very similar values to the ones it was expecting. The intended result for this fault type is for the drone to not move during the injection. As such, a distance deviation within the safe margin is expected paired with an increase in flight duration similar to the injection duration. The first two freeze campaigns, observable in A.1 and A.4, present similar results, with the second one even having less detected failures. The third campaign A.7 however, reliably presents a set of major failures - which include a run aborted by the flight controller. The flight controller always showed a strong resistance to this type of injection, and we believe that these campaigns solidify that, as the flight controller remains mostly unaffected for the two campaigns with smaller durations. Despite that, for a larger injection duration, the flight controller is not able to handle the values received and presents erratic behaviour, having a significant distance deviation. In figure 8.1 we present two logged trajectories, arbitrarily picked, from the freeze type injection runs. Trajectory A corresponds to a run whose fault injection lasted for 5 seconds, while trajectory B corresponds to a run whose fault injection lasted for 45 seconds. It is possible to observe that trajectory B presents a considerably higher deviation from the intended trajectory (figure 7.1) in comparison to the trajectory A.

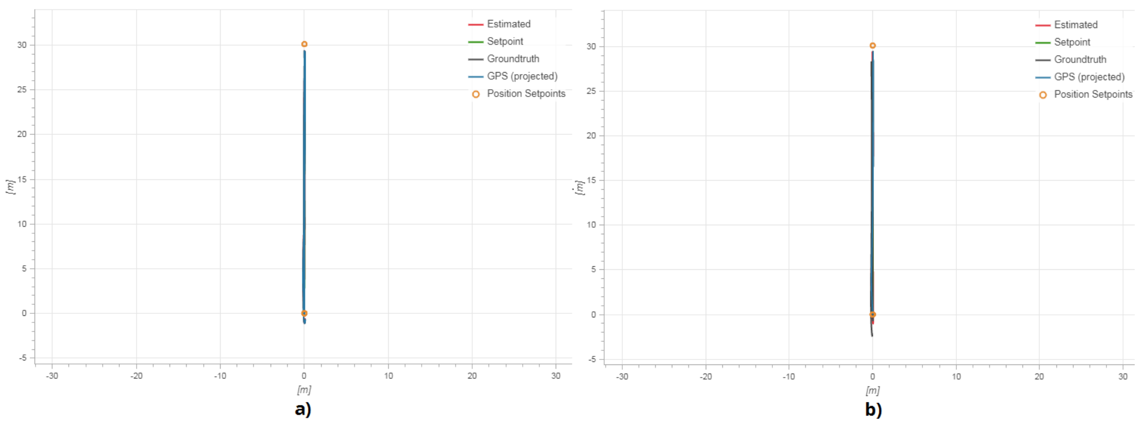


Figure 8.1: Freeze type logged trajectories: a) 5s Duration b) 45s Duration

For the **Delay** type injection, the results were as expected. Although we cannot observe a significant deviation from the planned flight route, its duration is significantly higher than the theoretical flight duration value - 60 seconds - for each of the delay type campaigns. When the message delay is active, a loss of communication between the sensors module and the position estimator module occurs, and subsequently, the drone stops for the duration of this delay. As previously explained, due to the fault injection interval using the sensors module's internal clock and the fault implementation causing a sleep-like effect in said module, the delay between messages is not taken into consideration by the fault injection interval. This causes the fault injection to last longer than what was previously planned.

This type was very effective, when taking the defined margins into consideration, seeing that for a 5 seconds injection duration (table A.3) every run was classified as a minor failure, and for a 15 and 45 seconds duration (tables A.6 and A.9) every run was considered a major failure, with the latter causing a run that had to be aborted, as there was a permanent loss of communication. As done with the previously addressed fault type, in figure 8.2 we present two logged trajectories, from the delay type injection runs, with trajectory A corresponding to a run whose fault injection lasted for 5 seconds, and trajectory B to a run whose fault injection lasted for 45 seconds. It is possible to observe that these trajectories present minor deviations from the intended mission. This is expected however, as this fault type was meant to target the flight's duration.

Lastly, for the **Fixed** type injection, these results were the hardest to analyze. Without a reliable reference of the drone's distance deviation, it is difficult to know exactly how much of the drone's trajectory was compromised. As this type is the one with the highest probability of affecting the drone's trajectory, being unable to know the exact trajectory might compromise our classification. We arbitrarily chose the coordinate equivalent to a latitude of 44 and a longitude of 44 for our fixed position.

When considering a fault injection duration of 5 seconds (table A.2), although most runs

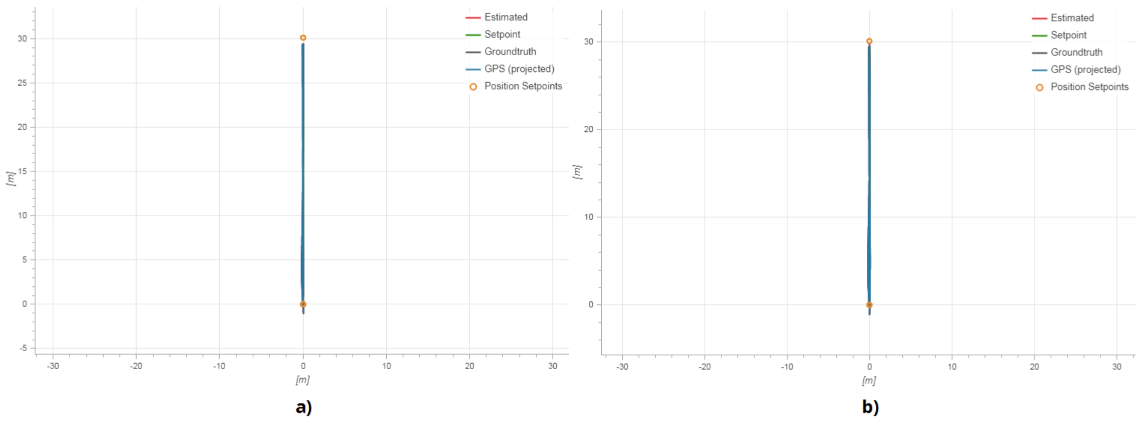


Figure 8.2: Delay type logged trajectories: a) 5s Duration b) 45s Duration

are classified as minor failures, the results are still relatively close to the reference values. However, when taking into consideration the results obtained for a 15 and 45 seconds duration (tables A.5 and A.8), the distance values registered in the log files round the 5000 kilometers. This is clearly not the actual distance covered by the drone, but since the flight duration values are considerably superior to the expected value and since we do not expect the fixed type injection to cause the drone to stay put, we assume that it caused the drone to significantly deviate from its course. As with the previous types, we present a figure (8.3) that showcases two of the logged trajectories for this injection type. Likewise, trajectory A corresponds to a run whose fault injection lasted for 5 seconds, and trajectory B to a run whose fault injection lasted for 45 seconds. Although we have already stated that the distances recorded by the log files are not entirely trustworthy, an interesting observation can be made using these two trajectories. For the flight presented in trajectory A, only the GPS projection is compromised while the estimated and setpoint trajectories are not. This causes a contrast to trajectory B, which presents both a compromised GPS projection and compromised estimated and setpoint trajectories. This observation, in conjunction with the higher flight duration values, validates our hypothesis that this fault type causes a significant deviation from the intended travelled distance.

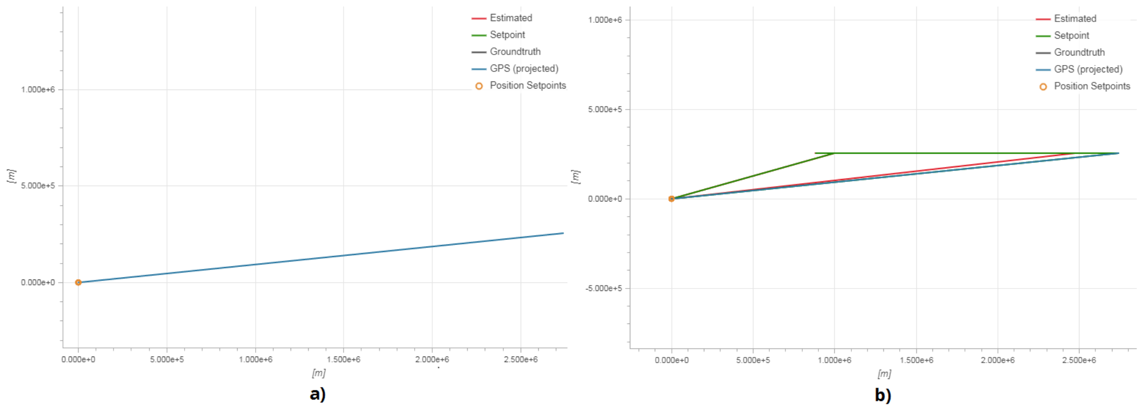


Figure 8.3: Fixed type logged trajectories: a) 5s Duration b) 45s Duration

8.2 Flight Controller Innate Failsafe

As already reported, PX4 contains various failsafes that allow it to detect anomalies within the system and abort flights that display said anomalies. For demonstrative purposes, we performed some experiments with the GPS Failsafe activation delay set to its minimum (1 second). In almost every case, a sudden change of value was met with an abort command that would cause the drone to immediately land. The only exception to this was the freeze type injection, which was able to remain active for, at most, 2 to 5 seconds. Although detecting an anomaly aborts the flight, we consider that issuing an instant landing order instead of returning the drone to the takeoff point can also be prejudicial, as the landing spot chosen might not safeguard the safety of the people or property around it.

Chapter 9

Conclusion

The past chapters demonstrate how this work advances the implementation of a unified tool for drone flight assessment. Although all objectives were met within an acceptable margin, we believe that the lack of clear requirements during the conceptualization phase possibly delayed our implementation - as some architectural conceptualization was done while implementing the tool. Even so, we believe that the previous chapters validate our tool's interface as well as the implemented faults. As the tool currently supports a flight controller that is available on commercial drones, it is an accessible way to assess said drones. Furthermore, since we implemented a script that converts BUBBLES coordinate files into drone missions, this tool can accomplish its role of providing the BUBBLES project with UAS flight trials in a controlled research environment.

Despite the fact that the objectives were met, a lot can still be done to further improve the tool. It is possible to expand it by either supporting other flight controllers, or by further increasing the array of available faults. Since we did not implement every fault present in our fault model, in the future it can be used as reference material to supplement the tool.

References

[1] MD Faiyaz Ahmed, Mohd Nayab Zafar, and JC Mohanta. Modeling and analysis of quadcopter f450 frame. In *2020 International Conference on Contemporary Computing and Applications (IC3A)*, pages 196–201. IEEE, 2020.

[2] L Meier. Px4 development guide. `https://dev.px4.io/master/en/`.

[3] Ferran Giones and Alexander Brem. From toys to tools: The co-evolution of technological and entrepreneurial developments in the drone industry. *Business Horizons*, 60(6):875–884, 2017.

[4] Sunghun Jung and Hyunsu Kim. Analysis of amazon prime air uav delivery service. *Journal of Knowledge Information Technology and Systems*, 12(2):253–266, 2017.

[5] Jessie YC Chen. Uav-guided navigation for ground robot tele-operation in a military reconnaissance environment. *Ergonomics*, 53(8):940–950, 2010.

[6] Janick Edinger, Dominik Schäfer, Christian Krupitzer, Vaskar Raychoudhury, and Christian Becker. Fault-avoidance strategies for context-aware schedulers in pervasive computing systems. In *2017 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 79–88. IEEE, 2017.

[7] Angus Stevenson. *Oxford dictionary of English*. Oxford University Press, USA, 2010.

[8] Abhishek Sharma, Pankhuri Vanjani, Nikhil Paliwal, Chathuranga M Wijerathna Basnayaka, Dushantha Nalin K Jayakody, Hwang-Cheng Wang, and P Muthuchidambaranathan. Communication and networking technologies for uavs: A survey. *Journal of Network and Computer Applications*, page 102739, 2020.

[9] UAS Task Force. Unmanned aircraft system airspace integration plan. *Department of Defense, March*, 2011.

[10] Fabio Ruggiero, Vincenzo Lippiello, and Anibal Ollero. Aerial manipulation: A literature review. *IEEE Robotics and Automation Letters*, 3(3):1957–1964, 2018.

[11] Gabriel Hoffmann, Haomiao Huang, Steven Waslander, and Claire Tomlin. Quadrotor helicopter flight dynamics and control: Theory and experiment. In *AIAA guidance, navigation and control conference and exhibit*, page 6461, 2007.

[12] M Farrukh Khan and Raymond A Paul. Pragmatic directions in engineering secure dependable systems. In *Advances in Computers*, volume 84, pages 141–167. Elsevier, 2012.

[13] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[14] Algirdas Avizienis. Fault-tolerance: The survival attribute of digital systems. *Proceedings of the IEEE*, 66(10):1109–1125, 1978.

[15] Jean-Claude Laprie. Dependable computing and fault-tolerance. *Digest of Papers FTCS-15*, pages 2–11, 1985.

[16] Ram Chillarege, Inderpal S Bhandari, Jarir K Chaar, Michael J Halliday, Diane S Moebus, Bonnie K Ray, and Man-Yuen Wong. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on software Engineering*, 18(11):943–956, 1992.

[17] Ethan M Puchaty and Daniel A DeLaurentis. A performance study of uav-based sensor networks under cyber attack. In *2011 6th International Conference on System of Systems Engineering*, pages 214–219. IEEE, 2011.

[18] Lav Gupta, Raj Jain, and Gabor Vaszkun. Survey of important issues in uav communication networks. *IEEE Communications Surveys & Tutorials*, 18(2):1123–1152, 2015.

[19] Goutam Kumar Saha. Software fault avoidance issues. *Ubiquity*, 2006(November):1–15, 2006.

[20] Harlan D Mills. The management of software engineering, part i: Principles of software engineering. *IBM Systems Journal*, 19(4):414–420, 1980.

[21] Haissam Ziade, Rafic A Ayoubi, Raoul Velazco, et al. A survey on fault injection techniques. *Int. Arab J. Inf. Technol.*, 1(2):171–186, 2004.

[22] Alfredo Benso and Paolo Prinetto. *Fault injection techniques and tools for embedded systems reliability evaluation*, volume 23. Springer Science & Business Media, 2003.

[23] Jens Guthoff and Volkmar Sieh. Combining software-implemented and simulation-based fault injection into a single fault injection method. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 196–206. IEEE, 1995.

[24] Reece Clothier and Rodney Walker. Determination and evaluation of uav safety objectives. In *Proceedings of the 21st International Conference on Unmanned Air Vehicle Systems*, pages 18–1. University of Bristol, 2006.

[25] Kay Wackwitz and Hendrick Boedecker. Safety risk assessment for uav operation. *Drone Industry Insights, Safe Airspace Integration Project, Part One, Hamburg, Germany*, 2015.

[26] Azza Allouch, Anis Koubaa, Mohamed Khalgui, and Tarek Abbes. Qualitative and quantitative risk analysis and safety assessment of unmanned aerial vehicles missions over the internet. *IEEE Access*, 7:53392–53410, 2019.

[27] Lawrence C Barr, Richard Newman, Ersin Ancel, Christine M Belcastro, John V Foster, Joni Evans, and David H Klyde. Preliminary risk assessment for small unmanned aircraft systems. In *17th AIAA Aviation Technology, Integration, and Operations Conference*, page 3272, 2017.

[28] Giorgio Guglieri, F Quagliotti, and Gianluca Ristorto. Operational issues and assessment of risk for light uavs. *Journal of Unmanned Vehicle Systems*, 2(4):119–129, 2014.

[29] Georg Macher, Eric Armengaud, Eugen Brenner, and Christian Kreiner. Threat and risk assessment methodologies in the automotive domain. *Procedia computer science*, 83:1288–1294, 2016.

[30] Kim Hartmann and Christoph Steup. The vulnerability of uavs to cyber attacks-an approach to the risk assessment. In *2013 5th international conference on cyber conflict (CYCON 2013)*, pages 1–23. IEEE, 2013.

[31] Eddy Deligne. Ardrone corruption. *Journal in Computer Virology*, 8(1-2):15–27, 2012.

[32] Michael Hooper, Yifan Tian, Runxuan Zhou, Bin Cao, Adrian P Lauf, Lanier Watkins, William H Robinson, and Wlajimir Alexis. Securing commercial wifi-based uavs from common security attacks. In *MILCOM 2016-2016 IEEE Military Communications Conference*, pages 1213–1218. IEEE, 2016.

[33] Joshua Gordon, Victoria Kraj, Ji Hun Hwang, and Ashok Raja. A security assessment for consumer wifi drones. In *2019 IEEE International Conference on Industrial Internet (ICII)*, pages 1–5. IEEE, 2019.

[34] Fekadu Lakew Yihunie, Aman Kumar Singh, and Sajal Bhatia. Assessing and exploiting security vulnerabilities of unmanned aerial vehicles. In *Smart Systems and IoT: Innovations in Computing*, pages 701–710. Springer, 2020.

[35] Vignesh Kumar Chandhrasekaran and Eunmi Choi. Fault tolerance system for uav using hardware in the loop simulation. In *4th International Conference on New Trends in Information Science and Service Science*, pages 293–300. IEEE, 2010.

[36] Daniel Mendes, Naghmeh Ivaki, and Henrique Madeira. Effects of gps spoofing on unmanned aerial vehicles. In *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 155–160. IEEE, 2018.

[37] Majd Saied, Benjamin Lussier, Isabelle Fantoni, Hassan Shraim, and Clovis Francis. Fault diagnosis and fault-tolerant control of an octorotor uav using motors speeds measurements. *IFAC-PapersOnLine*, 50(1):5263–5268, 2017.

[38] Qingqing Wu, Weidong Mei, and Rui Zhang. Safeguarding wireless network with uavs: A physical layer security perspective. *IEEE Wireless Communications*, 26(5):12–18, 2019.

[39] DroneKit Team. DroneKit Official Documentation. `https://dronekit-python.readthedocs.io/en/latest/about/overview.html` (Jun. 2021).

[40] Aicha Idriss Hentati, Lobna Krichen, Mohamed Fourati, and Lamia Chaari Fourati. Simulation tools, environments and frameworks for uav systems performance analysis. In *2018 14th International Wireless Communications & Mobile Computing Conference (IWCMC)*, pages 1495–1500. IEEE, 2018.

[41] Aakif Mairaj, Asif I Baba, and Ahmad Y Javaid. Application specific drone simulators: Recent advances and challenges. *Simulation Modelling Practice and Theory*, 94:100–117, 2019.

[42] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE, 2004.

[43] Johannes Meyer, Alexander Sendobry, Stefan Kohlbrecher, Uwe Klingauf, and Oskar Von Stryk. Comprehensive simulation of quadrotor uavs using ros and gazebo. In *International conference on simulation, modeling, and programming for autonomous robots*, pages 400–411. Springer, 2012.

[44] MAVLink Core Development Team. MAVLink File Format. `https://mavlink.io/en/file_formats/` (Jun. 2021).

[45] Gazebo Team. Gazebo Tutorials. `http://gazebosim.org/tutorials` (Jun. 2021).

[46] PX4 Team. PX4 Flight Review. `https://logs.px4.io/` (Jun. 2021).

[47] PX4 Team. PX4 Official GitHub. `https://github.com/PX4/` (Jun. 2021).

Appendices

This page is intentionally left blank.

Appendix A

Gathered Results

The following tables show the results of all the simulated runs.

Table A.1: GPS Freeze Injection Duration 5 Test Results

Injection Duration (s)	**Flight Duration (s)**	**Actual Flight Duration (s)**	**Maximum Deviation (m)**	**Classification**
5	72	78	4	Minor Failure
5	74	81	4,2	Minor Failure
5	68	87	3,7	Minor Failure
5	72	80	3,5	Minor Failure
5	69	83	3,5	Minor Failure
5	71	104	3,4	Minor Failure
5	73	80	3,9	Minor Failure
5	72	82	1,5	Normal
5	72	84	1,6	Normal
5	73	94	4	Minor Failure

Table A.2: GPS Fixed Injection Duration 5 Test Results

Injection Duration (s)	**Flight Duration (s)**	**Actual Flight Duration (s)**	**Registered Maximum Deviation (m)**	**Classification**
5	73	88	4,5	Minor Failure
5	68	95	4,5	Minor Failure
5	68	81	2,9	Normal
5	72	80	4,5	Minor Failure
5	72	82	4,3	Minor Failure
5	69	84	2,7	Normal
5	71	81	2,6	Normal
5	73	110	4,6	Minor Failure
5	73	80	4,1	Minor Failure
5	74	84	3,1	Normal

Table A.3: GPS Delay Injection Duration 5 Test Results

Injection Duration (s)	**Flight Duration (s)**	**Actual Flight Duration (s)**	**Maximum Deviation (m)**	**Classification**
5	67	144	3,4	Minor Failure
5	75	138	3,4	Minor Failure
5	70	155	4,1	Minor Failure
5	73	159	4,2	Minor Failure
5	67	155	4	Minor Failure
5	73	140	4,1	Minor Failure
5	72	153	3,4	Minor Failure
5	68	153	3,9	Minor Failure
5	72	157	4	Minor Failure
5	72	150	4,1	Minor Failure

Table A.4: GPS Freeze Injection Duration 15 Test Results

Injection Duration (s)	**Flight Duration (s)**	**Actual Flight Duration (s)**	**Maximum Deviation (m)**	**Classification**
15	73	83	3,3	Normal
15	76	104	4,1	Minor Failure
15	75	108	4,1	Minor Failure
15	73	85	2,7	Normal
15	73	86	4,2	Minor Failure
15	72	82	1,8	Normal
15	74	80	4,1	Minor Failure
15	73	84	2	Normal
15	68	82	4,3	Minor Failure
15	72	84	4,4	Minor Failure

Table A.5: GPS Fixed Injection Duration 15 Test Results

Injection Duration (s)	**Flight Duration (s)**	**Actual Flight Duration (s)**	**Maximum Deviation (m)**	**Classification**
15	118	134	5008150	Major Failure
15	156	169	5124270	Major Failure
15	152	184	5499950	Major Failure
15	126	179	5008020	Major Failure
15	122	153	5008030	Major Failure
15	111	168	4684700	Major Failure
15	95	136	4684660	Major Failure
15	126	135	5370070	Major Failure
15	111	154	4585160	Major Failure
15	134	138	5369910	Major Failure

Table A.6: GPS Delay Injection Duration 15 Test Results

Injection Duration (s)	**Flight Duration (s)**	**Actual Flight Duration (s)**	**Maximum Deviation (m)**	**Classification**
15	70	324	3,4	Major Failure
15	70	350	3,7	Major Failure
15	73	312	4,3	Major Failure
15	96	293	2,7	Major Failure
15	72	337	4	Major Failure
15	68	295	2,7	Major Failure
15	72	292	4,4	Major Failure
15	70	294	3,5	Major Failure
15	72	387	4	Major Failure
15	72	284	4	Major Failure

Table A.7: GPS Freeze Injection Duration 45 Test Results

Injection Duration (s)	**Flight Duration (s)**	**Actual Flight Duration (s)**	**Maximum Deviation (m)**	**Classification**
45	73	112	13,8	Major Failure
45	70	87	1,2	Normal
45	69	82	12,6	Major Failure
45	74	85	11,6	Major Failure
45	72	92	15,4	Major Failure
45	71	124	3,7	Minor Failure
45	N/A	N/A	N/A	Major Failure
45	69	93	14,9	Major Failure
45	95	85	12	Major Failure
45	69	126	15,4	Major Failure

Table A.8: GPS Fixed Injection Duration 45 Test Results

Injection Duration (s)	**Flight Duration (s)**	**Actual Flight Duration (s)**	**Maximum Deviation (m)**	**Classification**
45	162	197	13093790	Major Failure
45	188	200	11118120	Major Failure
45	203	241	12154760	Major Failure
45	202	256	11643800	Major Failure
45	127	187	11971130	Major Failure
45	253	312	11402280	Major Failure
45	205	257	11532460	Major Failure
45	164	211	11207410	Major Failure
45	196	196	11486610	Major Failure
45	166	189	11769640	Major Failure

Table A.9: GPS Delay Injection Duration 45 Test Results

Injection Duration (s)	**Flight Duration (s)**	**Actual Flight Duration (s)**	**Maximum Deviation (m)**	**Classification**
45	75	618	3,9	Major Failure
45	67	433	4,1	Major Failure
45	N/A	N/A	N/A	Major Failure
45	74	559	2,8	Major Failure
45	72	489	4,1	Major Failure
45	72	553	3,1	Major Failure
45	68	565	3,5	Major Failure
45	72	535	4	Major Failure
45	74	556	3	Major Failure
45	71	528	3,8	Major Failure