An Exploratory Study of Autopilot Software Bugs in Unmanned Aerial Vehicles

Dinghua Wang*
University of Technology Sydney, Australia, Southern University of Science and Technology, China
dinghua.wang@student.uts.edu.au

Shuqing Li
Southern University of Science and Technology, China
lisq2017@mail.sustech.edu.cn

Guanping Xiao
Nanjing University of Aeronautics and Astronautics, China
gpxiao@nuaa.edu.cn

Yepang Liu†
Guangdong Provincial Key Laboratory of Brain-inspired Intelligent Computation, Southern University of Science and Technology, China
liuyp1@sustech.edu.cn

Yulei Sui
University of Technology Sydney, Australia
yulei.sui@uts.edu.au

ABSTRACT

Unmanned aerial vehicles (UAVs) are becoming increasingly important and widely used in modern society. Software bugs in these systems can cause severe issues, such as system crashes, hangs, and undefined behaviors. Some bugs can also be exploited by hackers to launch security attacks, resulting in catastrophic consequences. Therefore, techniques that can help detect and fix software bugs in UAVs are highly desirable. However, although there are many existing studies on bugs in various types of software, the characteristics of UAV software bugs have never been systematically studied. This impedes the development of tools for assuring the dependability of UAVs. To bridge this gap, we conducted the first large-scale empirical study on two well-known open-source autopilot software platforms for UAVs, namely PX4 and Ardupilot, to characterize bugs in UAVs. Through analyzing 569 bugs from these two projects, we observed eight types of UAV-specific bugs (i.e., limit, math, inconsistency, priority, parameter, hardware support, correction, and initialization) and learned their root causes. Based on the bug taxonomy, we summarized common bug patterns and repairing strategies. We further identified five challenges associated with detecting and fixing such UAV-specific bugs. Our study can help researchers and practitioners to better understand the threats to the dependability of UAV systems and facilitate the future development of UAV bug diagnosis tools.

CCS CONCEPTS

• **Software and its engineering** → **Software defect analysis**; • **General and reference** → **Empirical studies**;

KEYWORDS

Unmanned aerial vehicles, software bugs, empirical study

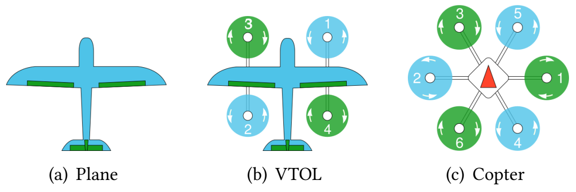


Figure 1: Example Drone Types Supported by PX4 [43].

1 INTRODUCTION

In recent years, the use of unmanned aerial vehicles (UAVs) is becoming more and more popular in daily life [54]. As a typical cyber-physical system (CPS), UAVs are context-aware. A UAV system perceives the external physical environment through various sensors and reacts in accordance with the external information [37, 47, 49]. There are three steps for a CPS to interact with physical environment, namely, sensing, decision-making, and action-taking [46].

Since UAVs are highly correlated with physical environment, the underlying software of a UAV differs significantly from traditional software, whose inputs are mostly stable and not interfered by external environment. On the one hand, the system inputs of a UAV can be dynamically fluctuating and uncontrollable with respect to the changes of environment. It is necessary for developers to consider the changes of environment proactively, making it challenging to build a reliable UAV system [54]. On the other hand, various configurations of sensors and hardware (e.g., the sensing range of a temperature sensor) should be taken into account during UAV's software development. Ignoring these configurations (e.g., parameters and limits) can cause reliability issues. In addition, some UAV systems are designed for more than one types of hardware. For example, as an open-source flight control software for autopilot, PX4[1] supports multiple types of drones, including Planes, VTOLs (Vertical Take-Off and Landings),and Copters, as shown in Fig. 1. Since different devices may have different functional limitations,

*Dinghua Wang is a student in the joint Ph.D. program of UTS and SUSTech.
†Yepang Liu is the corresponding author.
[1]https://px4.io/

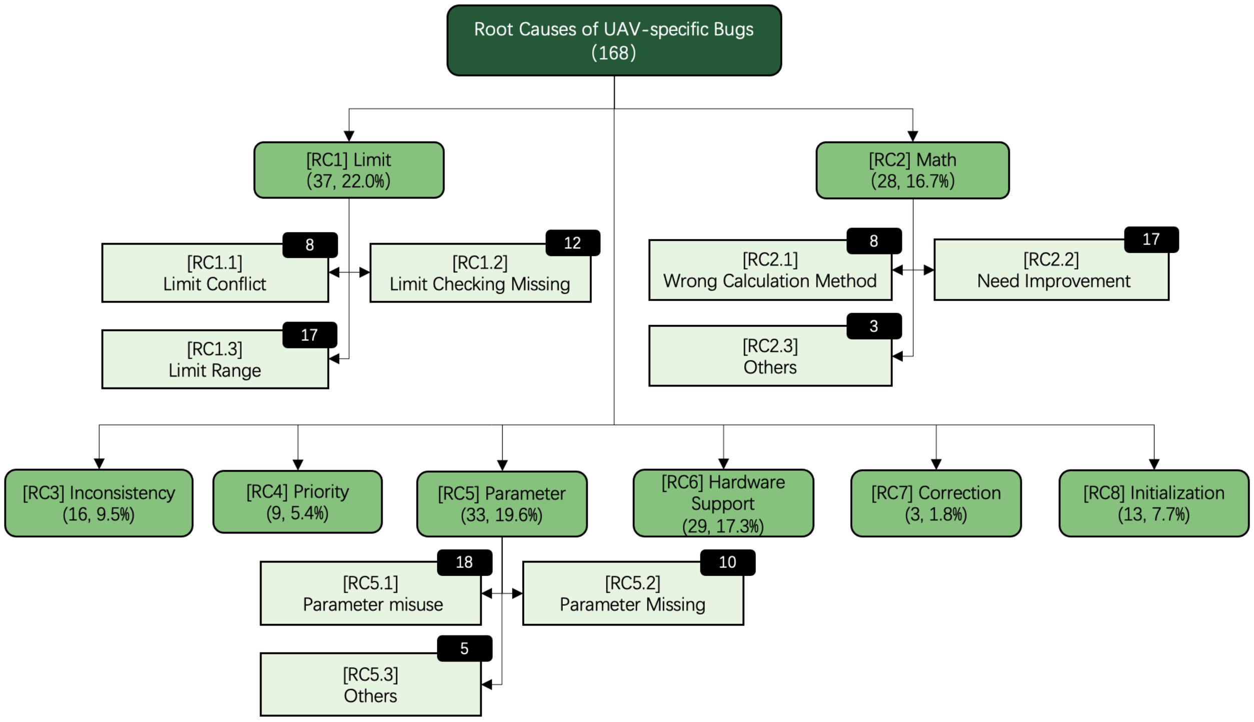

Root Causes of UAV-specific Bugs (168)
[RC1] Limit (37, 22.0%)
[RC1.1] Limit Conflict 8
[RC1.2] Limit Checking Missing 12
[RC1.3] Limit Range 17
[RC2] Math (28, 16.7%)
[RC2.1] Wrong Calculation Method 8
[RC2.2] Need Improvement 17
[RC2.3] Others 3
[RC3] Inconsistency (16, 9.5%)
[RC4] Priority (9, 5.4%)
[RC5] Parameter (33, 19.6%)
[RC5.1] Parameter misuse 18
[RC5.2] Parameter Missing 10
[RC5.3] Others 5
[RC6] Hardware Support (29, 17.3%)
[RC7] Correction (3, 1.8%)
[RC8] Initialization (13, 7.7%)

Figure 2: Root Causes of UAV-specific Bugs.

developers also need to consider the correspondence between different functions and hardware models to ensure the reliability of the whole system.

Although a lot of progress has been made in developing advanced UAVs, the reliability and safety of such systems are still major concerns, due to the aforementioned challenges [8]. Software bugs in UAV systems have caused property damage to the users [33]. Hence, there is an urgent need to analyze and understand the characteristics of bugs in UAV software in order to guide developers to build more reliable and secure UAV systems. Existing studies [32, 54] have explored some common bugs that might appear in both UAV software and traditional software, but they do not focus on UAV-specific bugs.

In this work, we present the first empirical study of UAV-specific bugs. Our study aims to provide practical yet systematic knowledge of UAV-specific bugs summarized and categorized from two major open-source autopilot software platforms (i.e., PX4 and Ardupilot) for drones. For our study, we collected 569 bugs from these two popular platforms on GitHub. Among those bugs, 168 are UAV-specific. We manually analyzed these 168 bugs by investigating their bug reports, source code, patches, and historical development data. Through our analyses, we observed eight types of root causes of UAV-specific bugs, as shown in Fig. 2. We have also summarized five challenges in detecting and fixing these bugs, as highlighted below.

- ***Challenge 1:*** It is difficult to design general test oracles to help locate bugs in UAV systems due to the unpredictability of system outputs and the need to evaluate system behavior from multiple perspectives.
- ***Challenge 2:*** Many bugs in UAV systems are difficult to reproduce in the presence of a dynamically changing physical environment.
- ***Challenge 3:*** Developing UAV software to support a variety of hardware often causes compatibility and dependence problems, which are hard to detect via code analysis.
- ***Challenge 4:*** Fixing low-level software bugs related to hardware and various system configurations is highly challenging.
- ***Challenge 5:*** Comparing the expected value with the actual value at a specific program point is a common method of fault localization. However, fault localization is generally difficult for UAV systems since a value or state at a specific program point is often unpredictable.

Our study can help researchers and developers to gain a better understanding of UAV-specific bugs and provide assistance for future UAV system development and research. To conclude, our paper makes the following contributions:

- We conduct the first empirical study of the root causes of UAV-specific bugs, which could assist future research on detecting and fixing software bugs in UAV systems.
- We make an analysis of the challenges in detecting and fixing UAV-specific bugs.

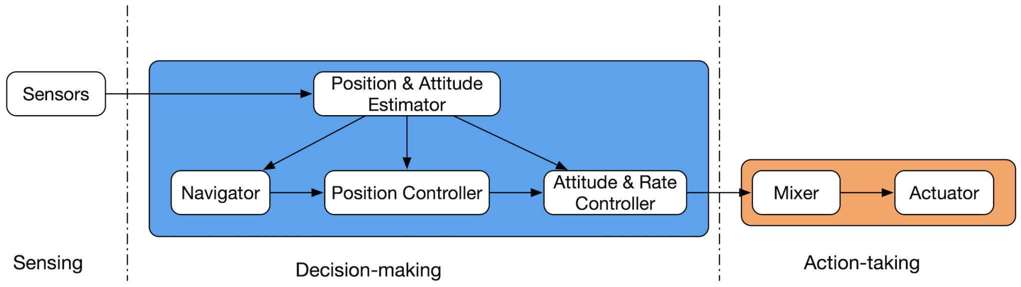


Figure 3: Flight Stack of PX4.

- We release the replication package and the dataset of UAV-specific bugs collected from PX4 and Ardupilot at https://doi.org/10.5281/zenodo.4898868.

The rest of the paper is organized as follows. Section 2 briefly gives the background knowledge of CPS dependability and the two examined projects (PX4 and Ardupilot). Section 3 describes the methodology of our empirical study, including the data collection and bug classification procedures. Section 4 presents the analysis of the root causes firstly and then presents the challenges of UAV-specific bugs detecting and fixing. Section 5 discusses our suggestions to practitioners. Section 6 summarizes the threats to validity, while Section 7 introduces the related work. Finally, Section 8 concludes the paper.

2 BACKGROUND

2.1 Cyber-Physical System Dependability

The term Cyber-physical system (CPS), widely referred in pervasive computing, was proposed in the 1990s by Weiser [55]. Recently, the development of CPSs has undergone considerable progress, especially in relation to drones, self-driving cars, and various IoT devices. However, developing and deploying a dependable CPS is still a difficult problem.

The three most important factors that affect the dependability of CPSs are context inconsistency, uncertainty, and faults in the system [54]. A CPS uses context to capture dynamic changes in the external physical environment. Context such as pressure, location, and temperature is captured by sensors. Context inconsistency means the inconsistency between the physical environment and the perceived context of the system, which is often manifested as environment noises. To explain conveniently, in our paper, we do not divide UAV systems into the physical system (a.k.a. the *plant* in control systems) and the controller when we talk about system input and output. Specifically, in our paper, the inputs of UAV systems involve the user input (e.g., command-line inputs) and the input from the physical environment. When we talk about the non-determinism problem of UAV system inputs, we are referring to the system input disturbances in the physical environment. UAV systems outputs in our paper are continuous physical trajectories [32].

2.2 PX4 and Ardupilot

Our study is performed on two popular open-source UAV platforms on GitHub, namely PX4 and Ardupilot. PX4 [44] is an open-source flight control system for drones and other unmanned vehicles. This is an active and well-maintained platform. It represents state of the art in the field of open source UAV system development. Fig. 3 shows the flight stack of PX4, which is a collection of guidance, navigation, and control algorithms for autonomous drones. As a typical CPS, PX4 consists of three subsystems: sensing, decision-making, and action-taking. We introduce several major components of the subsystems in the following. The estimator computes a vehicle state (mainly including position and attitude) using sensor inputs. The controller takes a setpoint from the navigator and an estimated state from the estimator as inputs. It controls the vehicle state until it matches the setpoint. For example, the position controller takes a position setpoint and an estimated position as inputs, and outputs the attitude and thrust setpoint to move the vehicle towards a desired position. The mixer translates the force commands into the individual motor commands and feeds motor commands to the actuator [40, 44, 53].

Ardupilot [3] is another open-source UAV platform for drones and unmanned vehicles. Ardupilot works on a wide variety of hardware to support multiple types of unmanned vehicles. It is constantly evolving based on rapid feedback from a large community of users and developers. The online community also provides massive bug data for our research.

3 METHODOLOGY

3.1 Data Collection

To obtain our initial taxonomy of UAV software bugs, we considered two popular and well-known open-source UAV systems, PX4 and Ardupilot. We did not choose other open-source projects, e.g., Openpilot [10] and Paparazzi [31], because they are not actively maintained and the number of issues is small. To collect bugs from the two projects, we defined the selection criteria consisting of the following rules:

Rule 1. The issue is closed. A closed issue means it has been resolved, therefore, such an issue report contains information that is helpful for bug classification and understanding.

Rule 2. The issue is labeled with *bug*. Developers usually label issue reports according to the characteristics of the reported issues.

Table 1: The Statistics of the Subjects Collected in Our Study

Project name	Stars	Commits	Lines of Code	Number of Files	Closed Issues	Bugs	UAV-specific Bugs
PX4	3,500	32,533	609,135	3,384	5,082	201	65
Ardupilot	4,900	42,826	1,505,836	3,291	3,478	368	103

A report with a *bug* label typically means that the described issue is caused by a system software defect, not other external factors.

Rule 3. There is a patch to fix the reported bug. The patch normally contains developers' comments and the code to fix the bug, which allows us to understand the root cause of bugs more easily.

Based on the above rules, we collected 569 real bugs out of 8,560 closed issues from the projects PX4 and Ardupilot on GitHub. Among them, PX4 contains 5,082 closed issues and 201 bugs, and Ardupilot has 3,478 closed issues and 368 bugs. As shown in Table 1, these two projects contain more than two million lines of code and over 70,000 commits.

3.2 Bug Classification

To characterize the root causes of the UAV-specific bugs, we analyzed the data by following the widely-adopted open coding procedure [7, 39]. Specifically, we performed an iterative manual labeling process that lasted half a year involving two annotators (i.e., co-authors of the paper), whom all have several years of CPS development experience. The iterations are as follows:

Iteration 1. For the 569 bugs, we analyzed three sources of data, i.e., bug reports (including comments from developers and users), commits, and bug patches on GitHub. According to our own understanding, we described and labeled each bug independently. We then compared the respective labeling results and analyzed those with large differences. With this iteration, we came up with a preliminary classification and labeling strategy.

Iteration 2. We independently labeled all bugs for a second round based on the preliminary classification and labeling strategy. We then compared the results and found that there were still a few differences in the labels. To resolve the differences, we discussed to clarify the boundaries between the labels. In the end, we slightly revised our bug classification and labeling strategy.

Iteration 3. We carried out a third iteration by going through all bugs following the revised classification and labeling strategy. Finally, we reached a consensus on the taxonomy of bug root causes.

After three iterations, each bug is labeled as one leaf category of our root cause taxonomy[2], which is shown in Fig. 2. As explained later, the leaf categories in our taxonomy are orthogonal, meaning that no bugs in our dataset can be classified into multiple categories from the perspective of root causes.

With the analysis of root causes, we identified 168 UAV-specific bugs from the 569 bugs following two rules. First, we considered a bug as UAV-specific if its root cause rarely exists in traditional software (e.g., bugs caused by the noisy physical environment). Second, if the root cause of a bug exists in traditional software but the bug has a much higher probability of appearing in UAV systems, we also considered it as UAV-specific. For example, the bugs in the category "hardware support" appear frequently in UAV systems due to the need to support many hardware platforms. For the characteristics of traditional software bugs, we referred to two previous empirical studies [52, 57] on bugs in Linux, Mozilla projects, and Apache projects. Due to the page limits, our subsequent discussion will focus on UAV-specific bugs.

[2]Note that this is not the only way to categorize the bugs in our dataset. Our current taxonomy is derived based on our knowledge and CPS development experience.

4 ROOT CAUSES AND CHALLENGES

With the identified UAV-specific bugs, we conducted an empirical study to understand the bugs' root causes and the challenges in dealing with these bugs. Specifically, we explore the following three research questions (RQs):

- **RQ1:** What are the common root causes of UAV-specific bugs?
- **RQ2:** What are the challenges in detecting UAV-specific bugs and how to address them?
- **RQ3:** What are the challenges of fixing UAV-specific bugs and how to address them?

By studying RQ1, we will understand how UAV-specific bugs arise. The root causes will guide the explanations of the findings for RQ2 and RQ3. Besides, in RQ2 and RQ3, we will investigate the new challenges imposed by the key differences between UAV systems and traditional software. After answering the three RQs, we also provide several suggestions to help developers build and debug UAV systems.

4.1 RQ1: Root Causes of Bugs

After conducting the manual classification described in Section 3.2, we observed that the root causes of the 168 UAV-specific bugs can be classified into eight categories: limit, math, inconsistency, priority, parameter, hardware support, correction, and initialization. Fig. 2 shows the number of bugs of these root causes. In the following, we will discuss each type of root causes in detail.

4.1.1 Root Cause 1: Limit. **Definition: limit bugs are those caused by improper parameter limits.** A UAV system is often compatible with a variety of different hardware, hence it is accompanied by a number of hardware limits. For example, PX4 has 1,306 limits [45], some of which are shown in Table 2. In practice, it is difficult for developers to handle a large number of limits correctly. When they make mistakes, the UAV systems may suffer from various types of limit bugs. In particular, we observed the following three subtypes of bugs of root cause 1:

- **Limit Conflict.** *Definition: developers set parameter limits that are in conflict with each other.* Due to various relationships between limits, two limits may have conflicts in their scope when set by developers. A typical example is issue #7097 [23] in PX4. In this issue, pitch limits are applied before applying setpoint offset,

Table 2: Example Limits of PX4

Name	Description	Min-Max	Default	Units
HDRIFT	Horizontal drift speed to use GPS	0.1-1.0	0.3	m/s
VDRIFT	Vertical drift speed to use GPS	0.1-1.5	0.5	m/s
EKF2_REQ_EPV	Required EPV to use GPS	2-100	8	m
FW_ACRO_Z_MAX	Acro body z max rate	10-180	45	degree
MT_FPA_MIN	Minimal flight path angle setpoint	-90.0-90.0	-20.0	degree

```
- radians(_parameters.pitch_limit_min),
- radians(_parameters.pitch_limit_max),
+ radians(_parameters.pitch_limit_min) - _parameters.
    pitchsp_offset_rad,
+ radians(_parameters.pitch_limit_max) - _parameters.
    pitchsp_offset_rad,
```

Listing 1: The Fix of Issue #7097.

which means that the pitch limit does not consider the offset of pitch setpoint. This conflict will cause the pitch limit value to increase by one unit of the setpoint offset. Listing 1 shows the patch to fix the bug #7097. To eliminate the conflict, the pitch limits need to subtract the offset.

- **Limit Checking Missing.** *Definition: developers miss the limit checking for the computation result.* Limits checking is frequently required in a UAV system, but developers may forget to check certain limits. In issue #7535 [24] of PX4, developers missed the limit checking of the setpoint after the velocity computation has been done. This issue caused the drone to fly away. Listing 2 shows that the developers added the constraints for all directions of the velocity setpoints to fix the bug.
- **Limit Range.** *Definition: the range of the limit is not right for some hardware or violates certain system requirements.* The range of some limits profoundly impacts the system behavior. Due to the lack of hardware knowledge, the limit range set by developers may compromise the stability of a UAV system. An example is issue #5305 [17] of PX4. The developers set the range of the actuator control parameter to be -1 to 1, which will enable the negative thrust. A right range should be 0 to 1. In Listing 3, the developers set the correct range to fix the problem of wing throttle because a drone with fixing wings will crash with a negative thrust.

4.1.2 Root Cause 2: Math. **Definition: math bugs arise from the misuse of mathematical formulas.** UAV systems often rely on various complex control and estimation algorithms. Comparing to traditional software, UAV software is more prone to math bugs. Sometimes, it can be very difficult for developers to accurately select the most suitable mathematical formula. In our dataset, there are two major subtypes of math bugs:

- **Wrong Calculation Method.** *Definition: developers use the wrong mathematical formulas/operations.* Developers use the wrong calculation formula, which can produce abnormal system outputs. In issue #8628 [29] of PX4, the developer incorrectly computed the mixing tables, which are a part of the output in mixer. As we mentioned in the background section, the mixer outputs the

```
+ // special velocity setpoint limitation for smooth
    takeoff (after slewrate!)
  if (_in_takeoff) {
      _in_takeoff = _takeoff_vel_limit < -_vel_sp(2);
      // ramp vertical velocity limit up to takeoff speed
-     _takeoff_vel_limit += _params.tko_speed * dt /
    _takeoff_ramp_time.get();
+     _takeoff_vel_limit += -_vel_sp(2) * dt /
    _takeoff_ramp_time.get();
      // limit vertical velocity to the current ramp
          value
      _vel_sp(2) = math::max(_vel_sp(2), -
          _takeoff_vel_limit);
  }

+ // make sure velocity setpoint is constrained in all
    directions (xyz)
+ float vel_norm_xy = sqrtf(_vel_sp(0) * _vel_sp(0) +
    _vel_sp(1) * _vel_sp(1));

+ if (vel_norm_xy > _vel_max_xy) {
+     _vel_sp(0) *= (_vel_max_xy / vel_norm_xy);
+     _vel_sp(1) *= (_vel_max_xy / vel_norm_xy);
+ }

+ _vel_sp(2) = math::constrain(_vel_sp(2), -_params.
    vel_max_up, _params.vel_max_down);
```

Listing 2: The Fix of Issue #7535.

value to the actuator. Due to this bug, the drone will exhibit abnormal flight status. Listing 4 shows the fix of issue #8628.

- **Need Improvement.** *Definition: the mathematical formulas used in program are imprecise or inefficient.* In some cases, there is no obvious error in the use of mathematical formulas or operations, but the calculation results are not accurate enough. This can be solved by using a better calculation method. For instance, as shown in Listing 5, developers fixed the issue #8198 [28] of PX4 by changing the old battery estimation algorithm to a more precise algorithm. Besides such cases, some math bugs are caused by inefficient computations and can be solved by fine-tuning the inefficient algorithm.

4.1.3 Root Cause 3: Inconsistency. **Definition: the root cause of the bugs is related to inconsistency between hardware and software.** Bugs often arise when developers are not familiar with the consistency between hardware and software in a UAV system. A typical case is that developers incorrectly use the function with a wrong drone models. Listing 6 gives an example [25]. The developers intended to use the function `land_detector` with multi-copter. Since `land_detector` is a function for another drone model `VTOL`, the drone cannot take off with this inconsistent function. In addition to the inconsistency between functions and drone models, there

```
+ /* fixed wing: scale throttle to 0..1 and other channels to -1..1 */
+ if (_actuators[index].output[i] > PWM_DEFAULT_MIN / 2) {
+     if (i != 3) {
+         /* scale PWM out PWM_DEFAULT_MIN..PWM_DEFAULT_MAX us to -1..1 for normal channels */
+         &msg.controls[i] = (_actuators[index].output[i] - pwm_center) / ((PWM_DEFAULT_MAX -
    PWM_DEFAULT_MIN) / 2);
+     } else {
+         /* scale PWM out PWM_DEFAULT_MIN..PWM_DEFAULT_MAX us to 0..1 for throttle */
+         msg.controls[i] = (_actuators[index].output[i] - PWM_DEFAULT_MIN) / (PWM_DEFAULT_MAX -
    PWM_DEFAULT_MIN);
+     }
+ }
```

Listing 3: The Fix of Issue #5305.

```
  float out = (roll * _rotors[i].roll_scale +
                  pitch * _rotors[i].pitch_scale) *
                      roll_pitch_scale +
                  yaw * _rotors[i].yaw_scale +
-                 thrust + boost;
+                 (thrust + boost) * _rotors[i].
     thrust_scale;
- out *= _rotors[i].out_scale;
```

Listing 4: The Fix of Issue #8628.

```
+ // The lower the voltage the more adjust the estimate
    with it to avoid deep discharge
+ const float weight_v = 3e-4f * (1 - _remaining_voltage)
    ;
+ _remaining = (1 - weight_v) * _remaining + weight_v *
    _remaining_voltage;
+ // directly apply current capacity slope calculated
    using current
+ _remaining -= _discharged_mah_loop / _capacity.get();
+ _remaining = math::max(_remaining, 0.f);
```

Listing 5: The Fix of Issue #8198.

```
  pwm_out_sim mode_pwm
  sensors start
  commander start
- land_detector start multicopter
+ land_detector start vtol
  navigator start
  ekf2 start
  vtol_att_control start
```

Listing 6: The Fix of Issue #7737.

```
- m_sensor_data.pressure_pa = convertPressure(
    pressure_from_sensor) / 256.0;
  m_sensor_data.temperature_c = convertTemperature(
      temperature_from_sensor) / 100.0;
+ m_sensor_data.pressure_pa = convertPressure(
    pressure_from_sensor) / 256.0;
  m_sensor_data.last_read_time_usec = DriverFramework::
      offsetTime();
  m_sensor_data.read_counter++;
```

Listing 7: The Fix of Issue #5243.

are also inconsistencies between hardware interfaces and interface protocols, inconsistencies between sensors and libraries, and so on. We observed that inconsistency bugs are more commonly found in the UAV system designed for multiple devices and may cause drone crashes when critical functions fail to work. Even worse, due to various complex and diverse inconsistencies between hardware and software, it could be very difficult for developers to completely avoid such bugs.

4.1.4 Root Cause 4: Priority. **Definition: the root cause of the bugs is related to hardware or software priority issues.** Unlike traditional software priority bugs, some of the UAV-specific priority bugs are caused by the hardware priority. These types of bugs are hard to be observed, because there are no obvious mistakes in the program logic and most such bugs do not cause significant deviation of system performance. Listing 7 gives a typical example of this type of bugs [19]. The patch which fixes the bug changes the order of pressure and temperature conversion. As we can see from the code snippet, the affected two statements have no common variables and it seems that such a fix would not cause any semantic changes. However, in many UAV systems, according to the bmp280 data sheet [50] and the sample pressure and temperature conversion code, the temperature conversion must be done before the pressure conversion, as the latter uses some results from the former [19]. In other words, there exists hidden data dependence between the two sensors. Such priority bugs require developers to have sufficient knowledge of the underlying hardware. Unfortunately, due to the diversity of hardware in UAV systems (e.g., different sensors and development boards), this requirement is often impractical for UAV software developers.

4.1.5 Root Cause 5: Parameter[3]. **Definition: the root cause of bugs is related to improper handling of parameters.** The parameters in a UAV system are very complicated. For instance, PX4 has more than 1,000 parameters. Most parameters include a limit and a default value, which are defined by developers based on the relevant function module attribute. As parameter settings affect the performance of a UAV system, improper handling of parameters may induce bugs. In our study, we found the following two major subtypes of root causes for parameter of bugs:

- **Parameter Missing.** *Definition: the parameter setting is not enough to meet the requirements of system functions.* Parameter setting is generally based on existing functional requirements. With the increase of system functions, if some relevant parameters are not set accordingly, the UAV system may exhibit unexpected behavior. In addition, the existing parameters may also be insufficient if developers fail to consider all possible scenarios when setting

[3]We did not put limit-related bugs in the root cause “parameter” because not all parameters have the limit property.

```
  _parameter_handles.land_slope_angle = param_find("
      FW_LND_ANG");
  _parameter_handles.land_H1_virt = param_find("
      FW_LND_HVIRT");
- _parameter_handles.land_flare_alt_relative = param_find
    ("FW_LND_FL_ALT");
+ _parameter_handles.land_flare_alt_relative = param_find
    ("FW_LND_FLALT");
  _parameter_handles.land_flare_pitch_min_deg =
      param_find("FW_LND_FL_PMIN");
  _parameter_handles.land_flare_pitch_max_deg =
      param_find("FW_LND_FL_PMAX");
  _parameter_handles.land_thrust_lim_alt_relative =
      param_find("FW_LND_TLALT");
```

Listing 8: The Fix of Issue #4745.

the parameters. In issue #6444 [20] of PX4, the drone started to plummet when the users wanted to nudge the drone during landing. To fix this bug, the developers added a parameter to allow users to operate the stick during landing.

- **Parameter Misuse.** *Definition: such bugs are caused by the misuse of certain parameters.* Since parameters have diversified characteristics, in order to use them correctly, developers need to understand the functions related to the parameters, the naming of the parameters, as well as the scope of the parameters. This is a non-trivial task and we observed various parameter misuses. For example, in issue #4745 [16] of PX4, the mistaken use of the two parameters FW_LND_FL_ALT and FW_LND_FLAT was caused by a similar naming. The patch to fix the bug is shown in Listing 8.

4.1.6 Root Cause 6: Hardware support. **Definition: the root cause of the bugs is related to the flawed support of certain hardware.** For this category, we only included the bugs related to the driver programs of a UAV's underlying hardware. There are also hardware support bugs in traditional software systems. We found that the hardware support bugs in UAV systems are no different to those in traditional systems. Most of them are caused by driver defects (e.g., compatibility issues). We call this type of bugs UAV-specific because the proportion of such bugs in a UAV system is large (i.e., 17.3%), as the UAV hardware support is generally not as good as in a traditional system (e.g., Linux).

4.1.7 Root Cause 7: Correction. **Definition: the root cause of the bugs is related to the correction of sensor data.** The data obtained by some sensors need to be corrected before they are used. For example, the GPS data can be disturbed by various environmental conditions (e.g., temperature) and therefore need to be corrected to ensure accuracy. Since intensive data corrections are required in a UAV system, developers may easily miss some correction process, resulting in various unexpected bugs.

4.1.8 Root Cause 8: Initialization. **Definition: the root cause of bugs is related to the initialization (or reset) of certain values.** Similar to data correction, we found that missing initialization is also a typical type of mistakes made by developers. Even some developers remember to do the initialization but they may often forget to redo the initialization (i.e., reset values) during the computation. Listing 9 shows an example where the developer forgets to initialize the battery during the calculation [30].

```
  AP_Notify::flags.pre_arm_check = true;
  AP_Notify::flags.pre_arm_gps_check = true;

+ // initialise battery
+ battery.init();

  // init baro before we start the GCS, so that the CLI
      baro test works
  barometer.set_log_baro_bit(MASK_LOG_IMU);
  barometer.init();
```

Listing 9: Pull Request #11208.

4.2 RQ2: Challenges in Bug Detection

To address RQ2, we first classified the UAV-specific bugs by answering the following two questions:

(1) Can the bugs be triggered by deterministic inputs?
(2) Can the bugs cause severe abnormal behavior (e.g., UAV crash, UAV unexpected trajectories)?

The first question concerns whether the bugs can be triggered easily, while the second question concerns whether the bugs can be observed or captured easily. If the answers to both questions are "Yes", the chances of detecting such bugs are relatively high. If the answers to both questions are "No", the bugs could be very difficult to detect. For RQ2, we focused on analyzing bugs with two "No" answers. Besides, we also analyzed the bugs with multiple pages of comments on GitHub because sometimes this means that developers need a lot of discussions to determine the location and cause of bugs before bug fixing. Based on the characteristics of UAV systems and the studied bugs, We identified three challenges for detecting UAV-specific bugs.

4.2.1 Challenge 1: Test Oracle Design. In traditional programs, an input usually corresponds to one or more specific outputs. This relationship can help check the correctness of a program [61]: when inputting a value, if the program outputs an unexpected state or value, the program is considered buggy. Here, the expected output of the program is a test oracle. It is known that designing an oracle for bug detection is a difficult problem when testing traditional software [4, 12, 14, 58]. The test oracle design is even harder for UAV systems, because the relations between the input and the output of a UAV is often non-deterministic. Typically, the input of a UAV can be divided into two parts: (1) the user input and (2) the input from the physical environment. The user input, such as the input from the drone's controller and the input from the command line interface of a simulator, is generally well defined. But the input from the physical environment is obtained by sensors. These inputs are highly dynamic and cannot be precisely defined or predicted. Although many developers have used parameter constraints to limit the range of the inputs from the physical environment, the vast volume of combinations of such inputs are still hard to be explored during testing. On the other hand, the UAV outputs generally contain continuous physical trajectories [32]. Given a command input to a drone, e.g., *fly from point A to point B*, if the drone outputs two different trajectories that both complete the task (as shown in Fig. 4), it would be hard to determine which trajectory is correct.

Furthermore, even if a drone outputs a physical trajectory that is exactly the same as in a previous successful test, the behavior

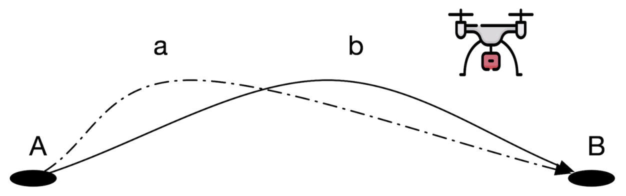


Figure 4: Which Trajectory (a or b) is Correct? [32]

can be wrong when other factors (e.g., wind speed) are considered. Because of this reason, to effectively test UAV systems, one has to holistically consider multiple factors. This is clearly challenging and it could be very difficult to determine an ideal oracle for UAV testing. In a recent study [32], the authors presented a trajectory prediction method to generate the test oracle for UAV systems. However, the application scenario of this method is limited to continuous and differentiable physical trajectories, and the correctness of the system is determined by simply judging whether the physical trajectory is smooth or not, without considering other factors.

By studying the bug comments, we found that UAV developers heavily relied on experts to judge the correctness of UAV outputs. As the judgment of the experts highly depends on their experience and domain knowledge, such human oracles may not be reliable. It is also difficult for experts to determine the correctness of UAV behaviors from one single output (e.g., trajectory). Future research may focus on the oracle problem of UAV testing and explore how to automate the testing process. We believe that it is also important to design testing frameworks that allow experts to clearly define criteria to judge the correctness of UAV behavior, so as to improve the reliability and reusability of human oracles.

4.2.2 Challenge 2: Bug Reproducing. Being able to reproduce a bug is crucial for verifying the correctness of bug detection in UAV systems [5, 6, 34]. The physical environment of UAV systems is non-deterministic, and it is possible that two executions behave differently in different physical environments. When reproducing a UAV-specific bug, the first step is typically to reproduce the running environment. However, the physical environment is highly dynamic and noisy, which means that it cannot be reproduced perfectly in the real world. Via our analysis of bug reports, we found that UAV developers mainly rely on simulators to reproduce bugs. Usually, a physical environment-level simulation is available and simple enough. Nonetheless, the simulation of drone systems is often simplified and unable to catch all the imperceptible changes (either system-level or environment-level) for bug detection. Implementing highly realistic simulation is expensive in terms of human effort and there are also many technical challenges (e.g., how to model and simulate uncertainty). Besides, the simulators may have bugs, which could bring new problems when reproducing UAV-specific bugs.

Traditional software also suffers from the non-determinism problem. However, in UAV systems, the problem becomes much more common because almost every execution could be affected by non-determinism. This problem needs more research attention.

4.2.3 Challenge 3: Hardware Dependence. Hardware dependence is a situation where there is data dependence between different hardware in a UAV system. While the data dependence problem in traditional software has been well explored, we observed that the UAV bugs caused by hardware dependence cannot be detected with the existing techniques. This is because some data dependence in UAV systems could be "hidden" in the hardware level. In the issue #5243 mentioned earlier, the developers triggered the bad performance of the drone. Because of hardware dependence, no bug can be found by analyzing the source code. This bug was discovered by a developer who was working on the bmp280 data sheet, which describes the hardware dependence between the pressure sensor and temperature sensor.

The computations in a UAV system are driven by the sensor inputs, which are limited by the parameter settings (see Table 2 for examples). However, the parameter settings rarely include the hardware dependence or the priority of sensors. Finding bugs caused by implicit hardware dependence could be a huge challenge for UAV software developers who lack knowledge of the underlying hardware. Future research may study how to address this challenge to ease bug finding in UAV systems.

4.3 RQ3: Challenges in Bug Fixing

To address RQ3, we first classified the UAV-specific bugs by answering the following questions:

(A) Is the bug fixed quickly (i.e., in a week)?
(B) Is the bug fixed with less than three patches?

Question (A) concerns the fixing time and question (B) concerns the difficulty of fixing. For RQ3, we then manually analyzed the bugs that have at least one "No" answer of the two questions (we call them *NA* and *NB* bugs) to find out the reason why developers took a long time to fix the bugs and why the bug fixing requires many (three or more) patches. With the analysis, we identified two challenges in fixing bugs in UAV systems.

4.3.1 Challenge 4: Hardware Related Bugs Fixing. Developers took a lot of time to fix some hardware related bugs. For example, 25 UAV-specific bugs in PX4 took developers more than one week to fix and 22 out of the 25 bugs are hardware-related bugs[4]. Bug fixing in a traditional program often aims to find solutions in the source code. However, in a UAV system, it is difficult for developers to find solutions from source code to fix hardware-related bugs. For example, in relation to issue #5243, which we have mentioned in Section 4.1.4, there is no obvious fault at the code level. Hence, it is hard for the developers to find a solution to such bugs in source code. Even if the developers know the properties of the hardware, it could still be difficult to fix such bugs. Due to the interdependence of hardware and software, unilateral compliance with hardware properties does not solve some bugs in UAV systems. For the issue #5305, the hardware parameter limits can be set to "-1 to 1" according to the hardware properties, but in practice, the drone's thrust cannot be set to a negative value. Therefore, when fixing such bugs, developers need to have both software and hardware knowledge, and also need to consider the usage scenarios of the system. What's

[4]These hardware-related bugs include parameter bugs, limit bugs, priority bugs, initialization bugs, correction bugs, and inconsistency bugs.

Table 3: Identified FL Strategies from PX4 UAV-specific Bugs

FL Strategy	Number of Bugs	Percentage
Flight Log	21	32.3%
Changing Parameters	9	13.8%
Bug Reproducing	3	4.6%
Changing Code	2	3.1%
Unknown	30	46.2%

more, considering that UAV systems also need to be compatible with different drone models, fixing hardware-related bugs could be more difficult. When attempting to fix issue #7746 [26] of PX4, the developers said that:

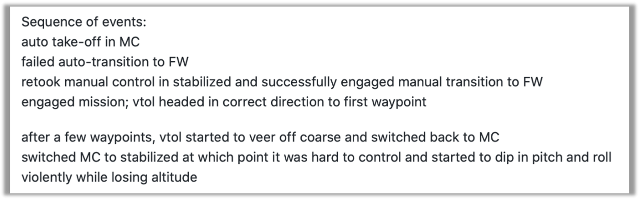
"However, for doing so I think I need to know more about FW and VTOL. Can you elaborate a bit more about vtol and landdetector? When is a vtol using MC and when FW? Why do we need to check for rotary wing in the MC landdetector (I thought that if we have a rotary wing, then we use MC landdector)?"

From the above comment, it can be inferred that the developer was confused about what strategies to take when fixing bugs for different drone models. In the future, it is highly desirable to design tools to give developers assistance to help them fix UAV-specific bugs more effectively.

4.3.2 *Challenge 5: Fault Localization.* Whether a bug is *NA* or *NB*, developers could spend a lot of time on fault localization. *NB* bugs have multiple fixing patches, which usually means multiple modules are involved. *NA* bugs contain many hardware-related bugs (e.g., 22/25 in PX4). All these have brought challenges to UAV fault localization, which is a critical step for debugging [35, 38, 56, 60]. A typical way to localize a fault in a program is to examine the program output at several program locations and compare the output values with the expected ones. However, as discussed earlier, in a UAV system, it is difficult for developers to predict the expected value/state at a certain program location. Although the expected values/states can be given by experts, it could still be difficult for developers to judge the correctness of UAV behavior due to non-determinism.

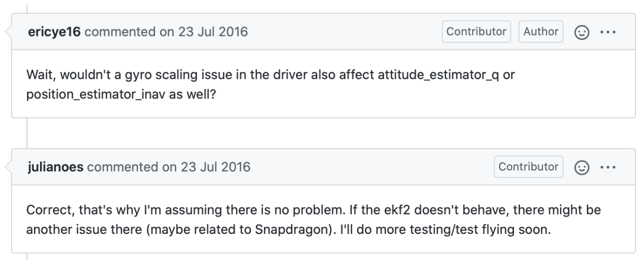
To understand how to overcome the challenge, we further analyzed 35 PX4 bugs, for which we can identify the developers' strategies of localizing faults by reading the bug reports. Interestingly, we observed four strategies that the developers often use to localize bugs, as summarized in Table 3. In the following, we discuss each of these strategies in detail.

Strategy 1: Referring to the Data in Multiple Flight Logs. A flight log often includes many details of the UAV system's runtime behavior. By inspecting multiple logs, developers can usually pinpoint the bug location more accurately. In issue #8186 [27] of PX4, the developer described a "priority" bug. The drone crashed after the sequence of events, as shown in Fig. 5. As we know, knowing the sequence of events that triggers a bug in an event-driven program can effectively help locate the faults. However, knowing the event sequence is not enough for fault localization in a UAV system. This is because unlike the clearly-defined events (e.g., clicking a button) in event-driven programs, UAV system events are usually vaguely described by developers or users. The description


Sequence of events:
auto take-off in MC
failed auto-transition to FW
retook manual control in stabilized and successfully engaged manual transition to FW
engaged mission; vtol headed in correct direction to first waypoint

after a few waypoints, vtol started to veer off coarse and switched back to MC
switched MC to stabilized at which point it was hard to control and started to dip in pitch and roll violently while losing altitude

Figure 5: Sequence of Events from Issue #8186 [27].


ericye16 commented on 23 Jul 2016

Wait, wouldn't a gyro scaling issue in the driver also affect attitude_estimator_q or position_estimator_inav as well?

julianoes commented on 23 Jul 2016

Correct, that's why I'm assuming there is no problem. If the ekf2 doesn't behave, there might be another issue there (maybe related to Snapdragon). I'll do more testing/test flying soon.

Figure 6: The Conversation Between the Bug Reporter and the Developer in PX4 Issue #5110 [18].

is often not detailed enough, making it difficult to locate the buggy modules and pinpoint the potential faults. In this case of issue #8186, the developer located the bug by examining the log of the inertial measurement unit (IMU) and the selected sensor. Through the abnormal timeout of the IMU and sensor, the developer located the fault at a system module (i.e., `sensors`), which can handle the sensor's timeout.

Strategy 2: Changing Parameters. In PX4 issue #5110 [18], the drone drifted when estimating attitude, and the reporter said:

"I've already tried changing parameters in Attitude Q estimator via QGroundControl but these changes don't seem to affect it."

As the comment indicates, a drone user or developer may change some related parameters (like the parameters in Attitude Q estimator in this case) to test the drone behavior and localize faulty modules, when the drone exhibits unexpected behavior. In the above case, although the system behavior was not influenced by the changes, changing parameters still guided the developer to locate the bugs, as shown in the Fig. 6.

Strategy 3: Bug Reproducing. Bug reproducing is a widely-used method to help locate faults in many software systems, but it is not observed in our studied UAV projects. In PX4, only three of the 35 bugs were located using bug reproducing. In issue #6669 [22] of PX4, the reporter reproduced the bug in the simulator, and all critical UAV runtime events are given to guide the fault localization. With the detailed bug report and the information obtained by reproducing the bug, the developers located and fixed the bug quickly. Although reproducing bugs can provide valuable information to help locate and fix the bugs, we found that, similar to many other projects, the data provided in many UAV bug reports are often very limited. To facilitate fault localization, UAV project maintainers may define standards to guide bug reporting (i.e., what information should be included).

Strategy 4: Changing Code. As UAV developers and users may not be professionals, when they localize faults, they may simply rely on code changing rather than using sophisticated techniques or tools. In PX4 issue 6651 [21], the UAV's takeoff could not be detected in altitude mode. The bug reporter tried to locate the fault by uncommenting some code lines to observe the UAV's behavior.

5 SUGGESTIONS FOR PRACTITIONERS

Based on our empirical analysis, we provide the following six suggestions for UAV practitioners.

- Developers should pay attention to the output from the controller or the underlying hardware. These outputs typically should be constrained within a reasonable range. For example, no matter what velocity or thrust computation has been performed, the output needs to fall within in a correct value range.
- Developers should also pay attention to the correction of sensor data. Although data correction may only slightly adjust certain values, it may significantly affect the dependability of the system. For example, temperature-sensitive sensors, GPS sensors, magnetic sensors, and pressure sensors are all strongly related to flight control. A small error in collecting and processing data from these sensors can cause serious consequences.
- Changing the values or properties of global parameters should be done very carefully. Even if the parameters cause bugs, it may not be desirable to directly changing the parameters. For example, if the range of a parameter `t` is from 0 to 100, and a bug can be fixed by allowing `t` to take a value beyond 100, it is preferred to fix this bug with new variables, like `a_max = t_max + 100`, instead of directly changing the range of `t`, which may cause problems when other parts of the system also use `t`.
- Developers should pay special attention to the initialization of hardware. Proper initialization of the underlying hardware could make a UAV system more stable.
- As mentioned in Section 4.3.2, flight logs can be very useful for fault localization. UAV users may include critical log data when reporting bugs to facilitate bug diagnosis. Nonetheless, it requires sufficient expertise and debugging experience to ensure the accuracy of fault localization via log analysis. In the future, it is desirable to design useful tools to help developers effectively analyze flight logs.
- Developers should be careful when using functions in a UAV system that supports multiple hardware models. For example, in PX4, most of the functions applicable to multi-rotor and VTOL do not support the plane model since the plane's flight style is very different from that of multi-rotor and VTOL.

6 THREATS TO VALIDITY

The validity of our study results may be subject to several threats.

Project Selection. First, Our empirical study only involved two open-source UAV projects. The collected bugs may not be representative and comprehensive. Hence, our findings may not be generalizable. In fact, we tried to find more projects, but most other UAV systems are either closed-source or do not have enough bug data for our analysis. We found that there are only five well-known and large open-source UAV systems on GitHub, and the two projects studied in our paper are the most active ones with a total of 8,560 closed issues. To mitigate the threat, we collected and thoroughly analyzed 168 UAV-specific bugs in the two projects. Future studies may investigate more projects to further understand the characteristics of UAV software bugs.

Bug Selection. Second, similar to the previous studies (e.g., [9]), we only collected the issues with the "bug" label from the two project for our research. While the developers and maintainers of the two projects have good labeling habits, it is still possible for us to miss some real bugs that do not have the "bug" label. In our future extension of this work, we will consider studying issues with other labels or no labels.

Manual Analysis of Bugs. In our work, all bugs are analyzed manually and it is inevitable that we could bring subjective judgment into the process. To reduce the threat, we had a lot of discussions when analyzing and classifying bugs, and we also tried to find references from the comments in bug reports to help build our taxonomy. It took two experienced UAV developers and two other co-authors six months to analyze and cross-validate the results. Our results are also online for public scrutiny.

7 RELATED WORK

Empirical Studies of CPSs and Robotics Software. Zheng et al. [62] conducted an empirical study of CPSs consisting of three parts: a literature review, an online survey, and interviews. They reviewed an empirical study of CPS developers and used the findings to highlight the key challenges in verification and validation in CPSs and presented a research roadmap to address these challenges. Unlike this study, we focus on the real-world bugs in CPSs.

Joshua et al. [13] conducted a comprehensive study of bugs in self-driving cars. They introduced 13 root causes, 20 bug symptoms, and 18 categories of software components that are often affected by those bugs. They distilled 16 findings from their study to guide future research. Yang et al. [59] summarized the state-of-the-art research results and identified the challenges of developing dependable CPSs. There are three challenges: context management, fault detection, and uncertainty handling. They found that the physical environment could strongly affect the dependability of a CPS. Different from these two studies, our study summarizes the different types UAV system bugs and the challenges in detecting and fixing these bugs.

Neshenko et al. [42] provided a comprehensive classification of recent surveys on IoT vulnerabilities and they also provided a unique taxonomy, including the features of IoT vulnerabilities, their attack vectors, impacts, and corresponding remediation methodologies. In their paper, they analyzed the flaws of IoT from the perspective of attack and defense. In comparison, our bug analysis is not restricted to the security perspective.

For robotics software, Anders et al. [11] characterized the dependency bugs in ROS and studied the pervasiveness and potential solutions of these bugs. Michel et al. [2] studied the energy-related practices in robotics software. Sotiropoulos et al. [51] studied the probability of immersing software in virtual scenarios to conduct simulation-based testing, and found that most of the robot navigation bugs can be found even in low-fidelity simulation. Our study shows that although there exist development challenges that are common to UAV and robotics software, developing UAV software may encounter multiple specific problems due to its unique features, e.g., hidden data dependence between different hardware.

Testing and Detecting UAV Bugs. Lucio et al. [48] proposed a UAV test platform by employing Matlab/Simulink to run the drones under test. Their paper mainly focuses on the control system, not software bugs. Claudio et al. [41] proposed an automated approach to generating an online test oracle for CPSs by identifying signal first-order logic (SFOL) fragments to specify requirements, defining the quantitative semantics of the fragment, and correctly converting the fragment into Simulink. Uttam et al. [1] presented a test bed for a cyber-physical power system. The test bed is built for hardware in loop simulation and system attacks, in order to generate data sets required by researchers. He et al. [32] proposed a system identification based oracle for fault localization in control-CPS software. They used the AR-SI algorithm to predict a physical trajectory which can be used as a test oracle for Ardupilot. The results show that the generated oracle increases the accuracy of CPS software fault localization. Kane et al. [36] proposed monitor based oracles for CPS testing and used an external runtime monitor as partial test oracles to detect unexpected CPS behavior. Giannakopoulo et al. [15] implemented a test case generation tool for autopilots. Symbolic execution has been used to generate both user inputs and test oracles. Our study is different from the aforementioned studies in that it is the first empirical study of real UAV-specific bugs by analyzing bugs that have been fixed in GitHub projects.

8 CONCLUSION

In this work, we conducted a large-scale empirical study to characterize UAV-specific bugs in two open-source UAV platforms, namely PX4 and Ardupilot. We identified 168 UAV-specific bugs from 569 real bugs in the two projects on GitHub. By analyzing these bugs, we proposed a taxonomy of UAV-specific bugs and summarized five challenges for detecting and fixing bugs in UAV systems. We believe this study can facilitate the development of UAV systems and guide future research in related areas. In the future, we plan to leverage our empirical findings to develop program analysis tools for detecting and fixing bugs in UAV systems.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China (Grant No. 2019YFE0198100), National Natural Science Foundation of China (Grant Nos. 61932021, 62002163, and 61802164), Natural Science Foundation of Jiangsu Province (Grant No. BK20200441), Guangdong Provincial Key Laboratory (Grant No. 2020B121201001), and Australian Research Grants (Grant No. DP210101348).

REFERENCES

[1] Uttam Adhikari, Thomas H Morris, and Shengyi Pan. 2014. A cyber-physical power system test bed for intrusion detection systems. In *2014 IEEE PES General Meeting| Conference & Exposition*. IEEE, 1–5. https://doi.org/10.1109/PESGM.2014.6939262

[2] Michel Albonico, Ivano Malavolta, Gustavo Pinto, Emitza Guzman, Katerina Chinnappan, and Patricia Lago. 2021. Mining Energy-Related Practices in Robotics Software. *CoRR* abs/2103.13762 (2021). arXiv:2103.13762 https://arxiv.org/abs/2103.13762

[3] Ardupilot.org. 2020. *Homepage of Ardupilot*. Retrieved May 25, 2020 from https://ardupilot.org

[4] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[5] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiß, Rahul Premraj, and Thomas Zimmermann. 2007. Quality of bug reports in eclipse. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*. 21–25. https://doi.org/10.1145/1328279.1328284

[6] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What makes a good bug report?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 308–318. https://doi.org/10.1145/1453101.1453146

[7] John W Creswell and Cheryl N Poth. 2016. *Qualitative inquiry and research design: Choosing among five approaches*. SAGE publications. https://doi.org/10.1177/1524839915580941

[8] Rogério De Lemos, Holger Giese, Hausi A Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M Villegas, Thomas Vogel, et al. 2013. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*. Springer, 1–32. https://doi.org/10.1007/978-3-642-35813-5_1

[9] Xiaoting Du, Guanping Xiao, and Yulei Sui. 2020. Fault Triggers in the TensorFlow Framework: An Experience Report. In *31st IEEE International Symposium on Software Reliability Engineering, ISSRE 2020, Coimbra, Portugal, October 12-15, 2020*, Marco Vieira, Henrique Madeira, Nuno Antunes, and Zheng Zheng (Eds.). IEEE, 1–12. https://doi.org/10.1109/ISSRE5003.2020.00010

[10] Emad Ebeid, Martin Skriver, Kristian Husum Terkildsen, Kjeld Jensen, and Ulrik Pagh Schultz. 2018. A survey of open-source UAV flight controllers and flight simulators. *Microprocessors and Microsystems* 61 (2018), 11–20. https://doi.org/10.1016/j.micpro.2018.05.002

[11] Anders Fischer-Nielsen, Zhoulai Fu, Ting Su, and Andrzej Wasowski. 2020. The forgotten case of the dependency bugs: on the example of the robot operating system. In *ICSE-SEIP 2020: 42nd International Conference on Software Engineering, Software Engineering in Practice, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 21–30.

[12] Gordon Fraser and Andreas Zeller. 2011. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering* 38, 2 (2011), 278–292. https://doi.org/10.1109/TSE.2011.93

[13] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Chen. 2020. A comprehensive study of autonomous vehicle bugs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 385–396. https://doi.org/10.1145/3377811.3380397

[14] Gregory Gay, Matt Staats, Michael Whalen, and Mats PE Heimdahl. 2015. Automated oracle data selection support. *IEEE Transactions on Software Engineering* 41, 11 (2015), 1119–1137. https://doi.org/10.1109/TSE.2015.2436920

[15] Dimitra Giannakopoulou, Neha Rungta, and Michael Feary. 2011. Automated test case generation for an autopilot requirement prototype. In *2011 IEEE International Conference on Systems, Man, and Cybernetics*. IEEE, 1825–1830. https://doi.org/10.1109/ICSMC.2011.6083936

[16] GitHub.com. 2020. *Issue 4745 of project PX4*. Retrieved May 25, 2020 from https://github.com/PX4/Firmware/issues/4745

[17] GitHub.com. 2020. *Issue 5035 of project PX4*. Retrieved May 25, 2020 from https://github.com/PX4/Firmware/issues/5305

[18] GitHub.com. 2020. *Issue 5110 of project PX4*. Retrieved May 25, 2020 from https://github.com/PX4/Firmware/issues/5110

[19] GitHub.com. 2020. *Issue 5243 of project PX4*. Retrieved May 25, 2020 from https://github.com/PX4/Firmware/issues/5243

[20] GitHub.com. 2020. *Issue 6444 of project PX4*. Retrieved May 25, 2020 from https://github.com/PX4/Firmware/issues/6444

[21] GitHub.com. 2020. *Issue 6651 of project PX4*. Retrieved May 25, 2020 from https://github.com/PX4/Firmware/issues/6651

[22] GitHub.com. 2020. *Issue 6669 of project PX4*. Retrieved May 25, 2020 from https://github.com/PX4/Firmware/issues/6669

[23] GitHub.com. 2020. *Issue 7079 of project PX4*. Retrieved May 25, 2020 from https://github.com/PX4/Firmware/issues/7097

[24] GitHub.com. 2020. *Issue 7535 of project PX4*. Retrieved May 25, 2020 from https://github.com/PX4/Firmware/issues/7535

[25] GitHub.com. 2020. *Issue 7737 of project PX4*. Retrieved May 25, 2020 from https://github.com/PX4/Firmware/issues/7737

[26] GitHub.com. 2020. *Issue 7746 of project PX4*. Retrieved May 25, 2020 from https://github.com/PX4/Firmware/issues/7746

[27] GitHub.com. 2020. *Issue 8186 of project PX4*. Retrieved May 25, 2020 from https://github.com/PX4/Firmware/issues/8186

[28] GitHub.com. 2020. *Issue 8189 of project PX4*. Retrieved May 25, 2020 from https://github.com/PX4/Firmware/issues/8189

[29] GitHub.com. 2020. *Issue 8628 of project PX4*. Retrieved May 25, 2020 from https://github.com/PX4/Firmware/issues/8628

[30] GitHub.com. 2020. *Pull Request 11208 of project Ardupilot*. Retrieved May 25, 2020 from https://github.com/ArduPilot/ardupilot/pull/11208

[31] Gautier Hattenberger, Murat Bronz, and Michel Gorraz. 2014. Using the paparazzi UAV system for scientific research. (2014).

[32] Zhijian He, Yao Chen, Enyan Huang, Qixin Wang, Yu Pei, and Haidong Yuan. 2019. A system identification based oracle for control-cps software fault localization. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 116–127. https://doi.org/10.1109/ICSE.2019.00029

[33] Lydia M. Hilton. 2017. *Personal Injury and Property Damage with Drones*. Retrieved Jan 25, 2021 from https://www.bfvlaw.com/personal-injury-and-property-damage-with-drones/

[34] Jeff Huang, Charles Zhang, and Julian Dolby. 2013. CLAP: recording local executions to reproduce concurrency failures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 141–152. https://doi.org/10.1145/2499370.2462167

[35] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 273–282. https://doi.org/10.1145/1101908.1101949

[36] Aaron Kane, Thomas Fuhrman, and Philip Koopman. 2014. Monitor based oracles for cyber-physical system testing: Practical experience report. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 148–155. https://doi.org/10.1109/DSN.2014.28

[37] Edward A Lee. 2008. Cyber physical systems: Design challenges. In *2008 11th IEEE international symposium on object and component-oriented real-time distributed computing (ISORC)*. IEEE, 363–369. https://doi.org/10.1109/ISORC.2008.25

[38] Ma łgorzata Steinder and Adarshpal S Sethi. 2004. A survey of fault localization techniques in computer networks. *Science of computer programming* 53, 2 (2004), 165–194. https://doi.org/10.1016/j.scico.2004.01.010

[39] Shuqing Li, Yechang Wu, Yi Liu, Dinghua Wang, Ming Wen, Yida Tao, Yulei Sui, and Yepang Liu. 2020. An Exploratory Study of Bugs in Extended Reality Applications on the Web. In *31st IEEE International Symposium on Software Reliability Engineering, ISSRE 2020, Coimbra, Portugal, October 12-15, 2020*, Marco Vieira, Henrique Madeira, Nuno Antunes, and Zheng Zheng (Eds.). IEEE, 172–183. https://doi.org/10.1109/ISSRE5003.2020.00025

[40] Lorenz Meier, Dominik Honegger, and Marc Pollefeys. 2015. PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE, 6235–6240. https://doi.org/10.1109/ICRA.2015.7140074

[41] Claudio Menghi, Shiva Nejati, Khouloud Gaaloul, and Lionel C Briand. 2019. Generating automated and online test oracles for simulink models with continuous and uncertain behaviors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 27–38. https://doi.org/10.1145/3338906.3338920

[42] Nataliia Neshenko, Elias Bou-Harb, Jorge Crichigno, Georges Kaddoum, and Nasir Ghani. 2019. Demystifying IoT security: an exhaustive survey on IoT vulnerabilities and a first empirical look on internet-scale IoT exploitations. *IEEE Communications Surveys & Tutorials* 21, 3 (2019), 2702–2733. https://doi.org/10.1109/COMST.2019.2910750

[43] PX4.io. 2020. *Airframe Reference in the User Guide of PX4*. Retrieved May 25, 2020 from https://docs.px4.io/master/en/airframes/airframe_reference.html

[44] PX4.io. 2020. *Flight Modes in the User Guide of PX4*. Retrieved May 25, 2020 from https://dev.px4.io/v1.9.0/en/concept/flight_modes.html

[45] PX4.io. 2020. *Parameter Reference in the User Guide of PX4*. Retrieved Jan 25, 2021 from https://docs.px4.io/master/zh/advanced/parameter_reference.html

[46] Yi Qin, Tao Xie, Chang Xu, Angello Astorga, and Jian Lu. 2019. CoMID: Context-Based Multiinvariant Detection for Monitoring Cyber-Physical Software. *IEEE Transactions on Reliability* 69, 1 (2019), 106–123. https://doi.org/10.1109/TR.2019.2933324

[47] Ragunathan Rajkumar, Insup Lee, Lui Sha, and John Stankovic. 2010. Cyber-physical systems: the next computing revolution. In *Design automation conference*. IEEE, 731–736. https://doi.org/10.1145/1837274.1837461

[48] Lucio R Ribeiro and Neusa Maria F Oliveira. 2010. UAV autopilot controllers test platform using Matlab/Simulink and X-Plane. In *2010 IEEE Frontiers in Education Conference (FIE)*. IEEE, S2H–1. https://doi.org/10.1109/FIE.2010.5673378

[49] Mahadev Satyanarayanan. 2001. Pervasive computing: Vision and challenges. *IEEE Personal communications* 8, 4 (2001), 10–17. https://doi.org/10.1109/98.943998

[50] Bosch Sensortec. 2015. Digital pressure sensor. *May 7th* (2015).

[51] Thierry Sotiropoulos, Hélène Waeselynck, Jérémie Guiochet, and Félix Ingrand. 2017. Can Robot Navigation Bugs Be Found in Simulation? An Exploratory Study. In *2017 IEEE International Conference on Software Quality, Reliability and Security, QRS 2017, Prague, Czech Republic, July 25-29, 2017*. IEEE, 150–159. https://doi.org/10.1109/QRS.2017.25

[52] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug characteristics in open source software. *Empirical software engineering* 19, 6 (2014), 1665–1705. https://doi.org/10.1007/s10664-013-9258-8

[53] PX4 Dev Team. 2017. Open Source for Drones-PX4 Pro Open Source Autopilot.

[54] N VANITHA and G PADMAVATHI. [n.d.]. Enhanced Security Mechanisms for the Selected Cyber Attacks in UAV Networks. ([n. d.]).

[55] Mark Weiser. 1999. The Computer for the 21<sup>st</sup> Century. *SIGMOBILE Mob. Comput. Commun. Rev.* 3, 3 (July 1999), 3–11. https://doi.org/10.1145/329124.329126

[56] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[57] Guanping Xiao, Zheng Zheng, Beibei Yin, Kishor S. Trivedi, Xiaoting Du, and Kai-Yuan Cai. 2019. An Empirical Study of Fault Triggers in the Linux Operating System: An Evolutionary Perspective. *IEEE Trans. Reliab.* 68, 4 (2019), 1356–1383. https://doi.org/10.1109/TR.2019.2916204

[58] Tao Xie. 2006. Augmenting automatically generated unit-test suites with regression oracle checking. In *European Conference on Object-Oriented Programming*. Springer, 380–403. https://doi.org/10.1007/11785477_23

[59] Wenhua Yang, Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2015. A survey on dependability improvement techniques for pervasive computing systems. *Science China Information Sciences* 58, 5 (2015), 1–14. https://doi.org/10.1007/s11432-015-5300-3

[60] Yanbing Yu, James Jones, and Mary Jean Harrold. 2008. An empirical study of the effects of test-suite reduction on fault localization. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 201–210. https://doi.org/10.1145/1368088.1368116

[61] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 129–140. https://doi.org/10.1145/3213846.3213866

[62] Xi Zheng and Christine Julien. 2015. Verification and validation in cyber physical systems: Research challenges and a way forward. In *2015 IEEE/ACM 1st International Workshop on Software Engineering for Smart Cyber-Physical Systems*. IEEE, 15–18. https://doi.org/10.1109/SEsCPS.2015.11