2021 International Conference on Unmanned Aircraft Systems (ICUAS)
Athens, Greece. June 15-18, 2021

A Study on Software Bugs in Unmanned Aircraft Systems

Max Taylor, Jayson Boubin, Haicheng Chen, Christopher Stewart, and Feng Qin

Abstract— **Control firmware in unmanned aircraft systems (UAS) manage the subsystems for in-flight dynamics, navigation and aircraft sensors. Computer systems on-board the aircraft and on gateway machines can now support rich features in the control firmware, such as GPS-driven waypoint missions and autonomy. However, the source code behind control firmware can harbor software bugs whose symptoms are detectable only during flight. Often, software bugs in UAS have serious symptoms that lead to dangerous situations. We studied previously reported bugs in the open-source repositories of ArduPilot and PX4, two widely used control firmware for UAS, and characterized their root causes, severity and position in the firmware architecture. Even though both platforms have employed rigorous software engineering practices, bugs were common and often had severe symptoms (e.g., crashes.) In particular, bugs associated with mishandling aircraft sensor readings were the leading cause for bug-induced crashes. Finally, we used simulation to study the symptoms of sensor bugs and found that source code repositories under reported their frequency and impact. Our study motivates multiple research directions on software reliability in UAS firmware.**

I. Introduction

Control firmware platforms, such as the ArduPilot Autopilot Software Suite (ArduPilot), fly unmanned aircraft systems (UAS) by sensing the state of the aircraft, responding to user commands, and adjusting pitch, roll, yaw, and thrust for navigation. In 2020, the code base for ArduPilot exceeded 700,000 lines of code [2] and libraries within the software suite added 2,000 lines of code per month [1]. Control firmware like ArduPilot now supports a wide range of features for UAS including: (1) support for multiple types of vehicles, (2) support for semi-autonomous flights between preset waypoint locations, (3) dynamic collision avoidance during flights, and (4) programmable interfaces that enable fully autonomous navigation [21], [22], [5]. Increasingly, UAS firmware need sophisticated computer systems, either on the aircraft or at nearby gateway machines. For example, the design of the Myriad processor was tailored for continuous image processing in UAS [4] and was used in the DJI Mavic, a popular UAS product.

Like all large software systems, UAS control firmware harbor bugs— i.e., mistakes in the source code that slip past compilers and software engineering tests and produce symptoms during flights. Often, bugs only produce symptoms if the aircraft, environment or control firmware encounter triggering conditions. The symptoms can vary depending on the bug, conditions, and environment. Benign symptoms may cause the aircraft to jerk during flight, before resuming normal flight. Severe symptoms include crashes and unresponsive aircraft. UAS crashes may not only destroy costly aircraft but also risk human lives, especially when unresponsive aircraft enter populated or restricted air space.

In this paper, we present an approach to study and characterize software bugs in widely used, open-source control firmware. Our approach uses online source code repositories and developer discussion forums to (1) find software patches intended to fix bugs in the code base, (2) understand the root cause of the bugs, and (3) characterize the severity of reported symptoms. Our analysis of reported software bugs quantifies prevalent root causes and symptoms and identifies areas within the code base that harbors severe bugs. We also study bugs under previously untested conditions to (1) determine if bug symptoms are reproducible, (2) quantify the scope of triggering conditions, and (3) measure the delay between triggering conditions and observed symptoms.

Our study examines 277 firmware bugs in the ArduPilot [2] and PX4 [19] code bases from 2016-2020. We reconstruct each bug using its corresponding software patch and developer comments and then label the root cause and symptoms. 21% of the firmware bugs in our study are severe, leading to crashes or unresponsive aircraft. Even though the root cause of most bugs is semantic (66%), bugs related to aircraft sensors are most likely to manifest severe symptoms. We re-insert sensor bugs into the code bases and simulate UAS flights across a wide range of conditions. We observe that the triggering conditions described in developer forums understated the full range of triggering conditions.

This paper presents a pragmatic approach to study software bugs in control firmware. By reconstructing root causes from code repositories, we uncover patterns in the root causes and symptoms of bugs in UAS firmware. Developers can use these patterns to preemptively search their code base for bugs. The results also call for renewed research on software reliability, e.g., verification and model checking, targeted at UAS. Specifically, the contributions of this paper are as follows:

1) We characterize 277 bugs in the ArduPilot and PX4 code repositories; the largest research study of these repositories to date.
2) We quantify the frequency of bugs by root causes, symptoms and location in the source code.
3) We define sensor bugs and explore their triggering conditions, presenting evidence that this class of bugs can produce severe symptoms more often than previously thought.

The rest of this paper is organized as follows:

2021 International Conference on Unmanned Aircraft Systems (ICUAS) | 978-1-6654-1535-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICUAS51884.2021.9476844

Authorized licensed use limited to: The Ohio State University. Downloaded on May 21,2023 at 02:06:41 UTC from IEEE Xplore. Restrictions apply.

- Section II compares our approach to prior bug studies.
- Section III describes how we form our bug collection.
- Section IV presents the main results of our bug study.
- Section V describes how to systematically reproduce sensor bugs.
- Section VI discusses our experience reproducing known sensor bugs.
- Section VII discusses the implications of our work.

II. Prior Work

This paper complements prior research that characterizes software bugs, i.e., bug studies. Bug studies target code bases that (1) have large developer communities and (2) underlie popular software products. The findings from bug studies can reduce the prevalence of bugs, improve automated tests, and lead to better programming paradigms. However, the findings from bug studies depend on available data. In particular, we compared prior work along the following dimensions:

1) Did the study cover all types of bugs or did it target certain types?
2) How many bugs were included?
3) Did the study characterize the runtime execution of bugs, i.e., were the bugs implemented and tested?
4) Did the study target unmanned aircraft systems?

Zhong et al. [27] comprehensively studied bugs and their corresponding patches in large Java projects, including Cassandra, Lucene and Mahout. The study covered 9,000 bugs, describing the root causes, location in code and efficacy of the patch. However, the patches were not tested directly under realistic conditions. Lu et al. [16] collected 105 concurrency bugs, i.e., mistakes in the source code that are triggered by the interaction among two or more threads of execution. While the study focused on a narrow class of bugs, it pioneered the data collection method used in our study, examining open-source repositories and codifying comments from developers. Chen et al. [6] studied 210 bugs in cloud systems and built a static analysis tool to detect common bugs. Garcia et al. [10] used this approach to comprehensively examine 250 bugs in autonomous vehicles.

Our work is closely related to bug studies that target performance bugs, i.e., mistakes in the source code that slow down system execution. Runtime analysis is critical to understand performance bugs. Jin et al. [14] examined 25 bugs from widely used server and desktop products, including Apache web server and OpenOffice. Stewart et al. [23] studied the runtime symptoms of 5 bugs in Java server applications, finding an approach to avoid triggering conditions during runtime.

Recent bug studies on unmanned aircraft systems (UAS) have focused on case studies (5-10 bugs). Timperly et al. [25] and Huang et al. [11] targeted semantic bugs, showing that simulation can faithfully reproduce bugs and that bug studies can provide insights for automated testing and patching procedures. Gungen et al. [1] used software engineering metrics to quantify the quality of the ArduPilot code base.

In comparison, this paper presents a comprehensive view of bugs in ArduPilot and PX4, two widely used control firmware for unmanned aircraft systems. Our analysis of 277 bugs covers every architectural component in the platforms. Further, we selected 5 bugs, reinserted them into the code base and studied their dynamic execution patterns at runtime. This allowed us to compare developer's comments about each bug's triggering conditions to actual triggering conditions observed during testing. We find that developers often underestimate the number of triggering conditions.

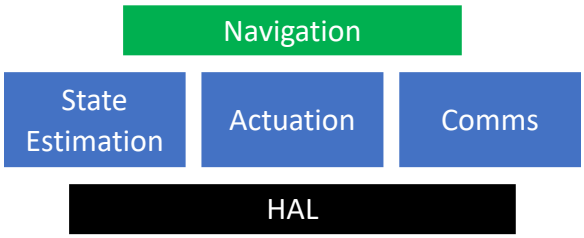

Fig. 1. Typical UAS firmware architecture.

III. Methodology

To ensure reliable operation in the presence of hardware failures, UAS come equipped with redundant hardware. However, the firmware responsible for managing hardware can contain bugs. In this case, redundant hardware offers no protection. This paper examines the effect of firmware bugs on UAS reliability.

UAS firmware manages all aspects of vehicle behavior. Figure 1 shows the typical architecture of UAS firmware. Firmware is typically divided into several subsystems:

- *Navigation* - sets waypoints and manages the UAS mission.
- *Actuation* - translates navigation commands to actuator controls.
- *State Estimation* - provides an estimate of the UAS physical state (e.g. position, orientation, etc.)
- *Communications* (Comms) - manages UAS communication with a ground control station (GCS).
- *Hardware Abstraction Layer* (HAL) - exposes a portable hardware API to other subsystems.

A. Target Firmware

We study two UAS firmware: PX4 and ArduPilot. We chose these firmware for four reasons. First, they have a large mindshare of developers (nearly 1,000 combined). These developers can gain immediate insight from our study. Second, PX4 and ArduPilot are stable. Their first releases appeared in 2009 and 2012. This allows us to learn what problems persist in UAS firmware. Third, commercial projects have adopted both firmware (e.g., [3]). Developers of other commercial UAS firmware can learn from the struggles of these firmware. Finally, both firmware are open source. Since all patches are public, we can review bugs the community has identified.

B. Bug Collection

Both firmware maintain issue trackers publicly available on Github. There have been more than 22,000 issues reported

Authorized licensed use limited to: The Ohio State University. Downloaded on May 21,2023 at 02:06:41 UTC from IEEE Xplore. Restrictions apply.

over their lifetimes. We apply several filtering rules to make this number manageable. First, we only consider issues reported between 2016-2020. This reduces the number of issues to just over 11,000. We also only consider issues closed during this period. This reduces the number to 3,266. Next, we require issues to have the "bug" label. This leaves us with 1,379 issues. We also require issues to have an accepted pull request (PR) associated with them. Note that not all issues are linked to PRs properly in these repositories, so we manually apply this filter. This leaves us with 489 issues.

We apply several manual filters to the remaining 489 issues. Some bugs are not in the UAS firmware, but instead are in general purpose libraries maintained by the projects. We remove those issues since they are not unique to UAS. We also remove bugs reported in the build system. Finally, we remove duplicate issues. This gives a final count of 277 bugs.

IV. Bug Study

This section presents the results of our bug study. We consider the following research questions:

- **RQ1**: *What types of bugs affect UAS firmware?*
- **RQ2**: *What are the symptoms of UAS firmware bugs?*
- **RQ3**: *How reproducible are UAS firmware bugs?*
- **RQ4**: *How do bugs affect each firmware component?*

We find that UAS bugs are dangerous: 21% have serious symptoms, such as crashes and fly-aways. Dangerous bugs are also likely to impact users: 74% do not depend on special hardware or timing conditions. Besides semantic bugs, code handling sensor events exhibits the most bugs (20%). These bugs are also more likely to be dangerous: about 34% have serious symptoms. Table I contains our main findings.

A. What types of bugs affect UAS firmware?

We categorize each bug as one of the following: *semantic*, *sensor*, *memory*, *concurrency*, or *other*. We define each type as:

- *Semantic* - bugs that do not misuse computational machinery, but violate user's expectations of UAS behavior.
- *Sensor* - illegal use of sensors (e.g. ignoring an IMU saturation flag.)
- *Memory* - illegal accesses to memory (e.g. reads from uninitialized memory and out of bounds accesses.)
- *Concurrency* - bugs that only occur in multithreaded environments (e.g. data races and deadlocks.)
- *Other* - includes code smells, syntax errors on main from incorrect merges, etc.

We choose this categorization for several reasons. First, memory and concurrency bugs are well-studied. Tooling exists to detect these bugs (e.g. Valgrind [18].) Second, the dependency on sensors to perceive the environment sets UAS firmware apart from other software. This motivates us to consider sensor bugs as a separate category. Finally, semantic bugs is a broad category characterized by what they are *not* - they do not misuse the onboard computer or avionics, like the other bugs, but instead are simply incorrect code. Since this category is so broad, it naturally captures many different behaviors. We break down the symptoms and add additional labels to bugs in this category to help make sense of them.

```
1   void handleMsg(MAVLinkMessage &msg) {
2     Location location;
3     switch (msg.coordinateFrame) {
4     ...
5     case MAV_FRAME_GLOBAL_INT:
6   -     loc.alt = msg.alt;
7   +     loc.alt = globalToRel(msg.alt);
8         loc.lat = msg.lat;
9         loc.lon = msg.lon;
10    }
11  }
```

Fig. 2. APM-3542: The MAVLink message handler supplies a value in the wrong coordinate frame to the navigation system.

1) Semantic Bugs: Semantic bugs dominate. Complex protocols contribute to this problem. For example, ground control stations (GCS) communicate with UAS using MAVLink [17]. MAVLink contains hundreds of commands. Commands often support several coordinate frames for providing parameters. Firmware message handlers must map the coordinate frames of each message to a common frame. But there is not a single common frame in ArduPilot or PX4; different components prefer different frames. To make matters worse, MAVLink does not standardize measurement units. Some messages use meters, others centimeters, etc. Message handlers must translate measurement units to the preferred system for the target component. Correctly supporting the semantics of MAVLink alone is a complex task.

Finding 1: Semantic bugs account for over 65% of bugs.

To further investigate this issue, we mark all bugs affecting MAVLink. We also label each bug where incorrect unit conversions play a factor. Nearly 11% of semantic bus involve MAVLink and over 10% involve incorrect unit conversions. Among all bugs, we find that 7% of all bugs involve MAVLink and 6.5% of all bugs involve incorrect units or orientation conversions.

Figure 2 is an example of a semantic bug. Here, ArduPilot's MAVLink message handler stores the message's altitude in the resulting location's altitude (line 6). When this code executes on a quadcopter, this is incorrect. ArduPilot's quadcopters navigate by using altitudes relative to the launch location's altitude. The navigation system interprets the altitude stored on line 6 as relative to the launch altitude, when it is in fact relative to sea level. This causes the navigation system to lead the UAS to an incorrect altitude. The fix is simple: convert the altitude to the correct coordinate frame (line 7.) Managing MAVLink is challenging. There are 21 different types of coordinate frames. As we see here, each vehicle requires custom handlers for each frame type. It is

Authorized licensed use limited to: The Ohio State University. Downloaded on May 21,2023 at 02:06:41 UTC from IEEE Xplore. Restrictions apply.

TABLE I

THE DISTRIBUTION OF THE ROOT CAUSES AND THE SYMPTOMS FOR BUGS IN ARDUPILOT AND PX4.

Root Cause	Serious Symptom		Not Serious Symptom				Total
	Crash	Fly-Away	Bad Navigation	Jerk	No Symptoms	Other	
Sensor	17	6	7	3	17	13	63
Memory	11	0	0	0	2	0	13
Concurrency	1	0	0	0	1	1	3
Semantic	18	6	39	7	81	32	183
Other	0	0	1	2	12	0	15
Total	47	12	47	12	113	46	277

easy for developers to make small mistakes like the one in Figure 2.

We identify three ways to address these challenges. We describe them from the easiest to implement to the hardest. First, developers and researchers can work together to create tooling to enforce their project's unit and coordinate frame conventions. This allows existing work to gradually improve. Second, developers can lean on their language's type system more. For example, each unit (e.g. meters) can have its own type. This way, the compiler can enforce that unit and coordinate frames are correct in new components. Finally, in the long-term, creating simpler communication protocols can ease the burden on developers. A simpler protocol should standardize coordinate frames and measurement units.

2) Sensor Bugs: Sensor bugs are the next largest share. One challenge faced by firmware is that navigation does not rely on sensor values directly. Instead, sensor readings are fused into a state prediction by an Extended Kalman Filter (EKF.) EKFs are complicated machines. They employ a model that uses observed variables (i.e. sensor readings) to form the most likely estimate of hidden variables (i.e. the UAS's physical orientation.) It is difficult to change the underlying model of the EKF when sensors fail, so firmware does not implement this. Instead, firmware relies on failsafes. But if multiple sensors experience faults, competing failsafe behaviors can cause undefined UAS behavior. Moreover, the state of the EKF becomes tainted if it uses stale values from the failed sensor instance.

Finding 2: Incorrect use of sensor values occurs in over 22% of bugs.

Figure 3 shows an example of a sensor bug. The navigation system checks the GPS speed (line 2.) If the speed is less than the stall speed, the navigation system invokes a crash handler (line 4.) However, this logic is incorrect. If the GPS fails, then the GPS speed is reported as 0. This causes the navigation system to incorrectly detect a stall, even though the UAS is still airborne. The crash handler proceeds to disarm the UAS, causing a crash. The solution is to verify the health of the GPS before using its measurement (line 3.)

3) Memory Bugs: Memory bugs (5% of bugs) occur more rarely in UAS firmware than other software (e.g., 12-16% in conventional software, found in [15]). We believe there are two main reasons. First, firmware developers deliberately

```
1  void doNavigation() {
2  - if (gps.speed() < STALL)
3  + if (gps.ok() && gps.speed() < STALL)
4        handleCrash();
5    ...
6  }
```

Fig. 3. APM-9349: The navigation system incorrectly uses a stale sensor value, causing the UAS to crash.

avoid memory allocation as much as possible. Allocation is often slow and energy-intensive. Fewer allocations presents fewer opportunities for programming mistakes. Second, PX4 and ArduPilot both use the Valgrind memory checker. This helps developers catch bugs before they can appear in stable releases.

Finding 3: Memory and concurrency bugs are rarer in UAS firmware than other software.

Despite the fact that memory bugs are rarer, they are more catastrophic in UAS firmware than other software. An invalid access in most software results in a relatively harmless program crash. However, when UAS firmware crashes the results are severe. Moreover, firmware supports environments without memory management units. Invalid accesses can silently corrupt the firmware's address space in these environments, leading to incorrect execution later on. Motivated attackers can target these bugs to hijack firmware control flow.

4) Concurrency Bugs: Concurrency bugs are also rare in UAS firmware, accounting for less than 1% of bugs. This is because both projects use threads in simple, consistent ways. PX4 runs each module in its own thread. Inter-thread communication uses an asynchronous publish/subscribe system. Since the flight tasks guiding UAS behavior must tolerate measurement uncertainty, strict delivery time guarantees are unnecessary. The use of the pub/sub system prevents data races. Asynchronous communication generally prevents blocking on messages from other components, making deadlocks rare.

ArduPilot uses a different approach for managing firmware tasks. Every module registers its tasks with the scheduler. A 1KHz timer invokes tasks according to their schedule on a single thread. ArduPilot only runs a few threads (e.g. several

Authorized licensed use limited to: The Ohio State University. Downloaded on May 21,2023 at 02:06:41 UTC from IEEE Xplore. Restrictions apply.

TABLE II
THE DISTRIBUTION OF THE ROOT CAUSES AND THE SYMPTOMS FOR BUGS IN EACH FIRMWARE COMPONENT.

Comp.	Serious Symptom		Not Serious Symptom				Total
	Crash	Fly-Away	Bad Navigation	Jerk	None	Other	
HAL	23	0	2	1	27	18	71
State Estimation	4	7	10	4	18	4	47
Actuation	4	2	10	3	5	5	29
Communication	1	1	3	0	22	8	35
Navigation	11	2	22	3	19	9	66

for I/O and one for the task system.) This approach reduces the probability of data races. Concurrent variable access is impossible for module tasks. Only the core background threads can contain data races.

B. What are the symptoms of UAS firmware bugs?

We manually label each bug's symptoms as one of the following:

- *Jerk* - when the UAS undergoes a transient position change. Typically, the pilot experiences the vehicle's position jumping and then stabilizing.
- *Crash* - when the UAS collides with an obstacle.
- *BadNav* - when the UAS navigates incorrectly (but the pilot can intervene.) This category includes issues such as position drift.
- *FlyAway* - when the pilot loses control of the UAS's behavior.
- *Other* - minor symptoms, ranging from incorrect log messages appearing to LEDs not properly lighting.

Finding 4: Over 21% of UAS bugs are dangerous.

We call a bug *dangerous* if its symptoms include crashes or flyaways. While incorrectly navigating or transient navigation noise can be dangerous, in both cases a pilot can intervene to prevent the situation from escalating further. But in the cases of crashes and flyaways, the pilot is unable to intervene to protect the vehicle and bystanders. For instance, the bug described in Figure 3 is dangerous because the pilot is unable to prevent a crash after a GPS loss. However, in the case of the bug in Figure 2, the pilot can resume manual control as the UAS navigates to the incorrect altitude.

Finding 5: 39% of dangerous bugs are caused by mishandling sensors.

Sensor bugs are the dominant cause of unsafe conditions in UAS. Firmware spends much time executing code that reads and transforms sensor values. Misusing sensor values (e.g. ignoring IMU saturation flags, using a value that is very stale, etc) often leads to severe consequences. The firmware's perception of the vehicle's state becomes warped, leading to incorrect actuation. A single bad moment of actuation can put the vehicle into a dangerous position that is difficult for the firmware (or human pilot) to correct.

C. How reproducible are UAS firmware bugs?

By understanding how reproducible UAS firmware bugs are, we can understand how likely users are to be affected by these bugs. We identify three levels of reproduction. The easiest bugs to reproduce do not require special hardware, settings, or triggering conditions. Harder bugs to reproduce require specific settings and triggering conditions. The hardest bugs to reproduce require specific hardware.

Finding 6: 74% of bugs are reproducible under default settings, 20% are reproducible under modified settings, and 6% are reproducible only on specific hardware or have special timing conditions.

We find that the majority of bugs are easily reproducible. This shows that simulation is a valuable tool for firmware developers.

Finding 7: Nearly 70% of dangerous bugs are reproducible under default settings with no special hardware or timing conditions.

Dangerous bugs are usually easy to reproduce. This suggests that thorough testing can help eliminate a significant portion of dangerous bugs in UAS.

D. How do bugs affect each firmware subsystem?

We label each bug with the subsystem containing the bug. Some bugs are caused by the interaction of several subsystems. We ignore these complex bugs, i.e. they are unlabeled in our data set. Other bugs are found in common libraries shared between different subsystems. We also ignore these bugs in this section. Interestingly, most bugs are contained entirely in a single subsystem: only 29 bugs are excluded by this filtering criteria. Our main findings are shown in Table II.

The HAL and navigation subsystems contain the most bugs. Nearly 10% of bugs in each subsystem involve incorrect unit types or coordinate frames. This highlights the challenge of providing consistent views of diverse hardware types. We notice that the HAL is the only subsystem affected by concurrency bugs. This is consistent with our discussion on ArduPilot and PX4's thread management policies. Most I/O originates from the HAL, and I/O is performed by a dedicated thread. Moreover, half of the memory bugs reside in the HAL. Finally, bugs in the HAL are more likely to be dangerous than the ones in other subsystems.

Authorized licensed use limited to: The Ohio State University. Downloaded on May 21,2023 at 02:06:41 UTC from IEEE Xplore. Restrictions apply.

V. Simulating Sensor Bugs

Our findings show that sensor bugs (1) are common in UAS (finding 2), (2) are the leading cause of serious symptoms in UAS (finding 5), and (3) are easy to reproduce. This motivates us to understand sensor bugs further. This section analyzes the triggering conditions of sensor bugs and describes fault injection systems to trigger sensor bugs.

A. How are sensor bugs triggered?

First, we look at how sensor bugs are triggered. We identify several common triggering conditions: *Software-Triggered*, *Measurement Errors*, *Failures*, *Healing*, and *Interrupt Storms*. Our key finding is that over 75% of sensor bugs are software-triggered or are triggered by total sensor failures, and have weak timing constraints to trigger the bug. Here, we explain each type of triggering condition.

Finding 8: Nearly 37% of sensor bugs affect vehicles under all workloads.

1) Software-Triggered: Sensor bugs are sometimes triggered by defects in control firmware. These bugs are triggered by normal use of the UAS, e.g. merely turning on the system triggers the bug. We find that 23 of the 63 sensor bugs in our collection have this triggering condition. Fortunately, only 4 of these bugs have dangerous consequences. Often, these bugs render the UAS inoperable. 17 of these 23 bugs can only be recreated on specific hardware because they reside in drivers. These bugs are hard to detect because they are hard to reproduce. Subtle programming errors can break sensor drivers. There is no technology available to simulate all the hardware used in the wild. These mistakes cannot be caught until the firmware is tested on real hardware.

Finding 9: Around 2% of UAS bugs are triggered by single measurement errors.

2) Measurement Errors: Some sensor bugs are triggered if the sensor instance reports an erroneous measurement. Our bug collection contains 6 sensor bugs triggered under measurement errors. Of these errors, a single bitflip in the sensor reading is enough to trigger 2 of these bugs. Recent work (e.g. [12]) attempts to address this category of bugs in autonomous vehicles.

Finding 10: Sensor failures are the dominant triggering condition of sensor bugs, accounting for nearly 40% of sensor bugs.

3) Failures: UAS firmware must correctly handle sensor instances failing. Unfortunately, firmware does not always handle these situations correctly. Of the 63 sensor bugs in our collection, 25 are triggered by sensor failures. Only 6 sensor bugs triggered by sensor failures need specific hardware to reproduce. Sensor failures are both the dominant triggering condition of sensor bugs and testable. It is worth noting that sensor bugs triggered by sensor failures represent a larger share of bugs in UAS than memory and concurrency bugs, a dominant bug category in other software [15].

```
1   void batteryFailsafe() {
2     switch (lowBatteryAction) {
3     case LOW_BAT_ACTION::RETURN_OR_LAND:
4       if (!modeTransition(AUTO_RTL))
5   -     warn("Can't execute failsafe");
6   +     forceDescend();
7       break;
8     ...
9     }
10  }
```

Fig. 4. PX4-13291: A dangerous bug with strong timing constraints.

To better understand this category, we look at the timing constraints needed to trigger sensor bugs caused by failures. We label each bug in this category as having weak, medium, or strong timing constraints. We define each label as:

- *Weak Timing Constraints* - Sensor bugs triggered by a failure at any time in any flight mode have weak timing constraints. This is the most common timing constraint, accounting for 16 of the 25 sensor bugs triggered by sensor failures.
- *Medium Timing Constraints* - Sensor bugs that can only be triggered by a single failure in a specific flight mode have medium timing constraints. We find that this is the third most common timing constraint, representing 4 of the 25 bugs in this category.
- *Strong Timing Constraints* - Sensor bugs that can only be triggered by a sequence of failures in specific flight modes have strong timing constraints. We find that 5 of the 25 sensor bugs share this constraint.

Finding 11: The majority (64%) of sensor bugs triggered by sensor failures have weak timing constraints.

Figure 4 shows an example of a bug in PX4 with strong timing constraints. To trigger this bug, the UAS first must experience a GPS failure. This causes the firmware to lose its position estimate. The firmware executes a user-configured fail-safe when the position estimate is lost. While the fail-safe executes, the battery becomes critically low. This causes the firmware to invoke its battery fail-safe procedure (line 1 of the code listing.) The battery fail-safe attempts to execute a return to launch or land (line 5.) However, since no position estimate is available, the fail-safe execution fails. Instead, the vehicle enters a loiter mode, cancelling the previous fail-safe. Since the battery is at a critically low level, the UAS quickly loses power and crashes. The fix is to force the UAS to descend if it is unable to execute its battery fail-safe (line 6.) The timing constraints are strong for this bug because multiple sensors must fail in a specific sequence. If the GPS does not fail soon before the battery fail-safe, then the bug is not triggered.

4) Healing: Sometimes a failed sensor triggers a bug if the sensor becomes healthy again. We say that these bugs are triggered by sensor *healing*. We find that these bugs are rare,

Authorized licensed use limited to: The Ohio State University. Downloaded on May 21,2023 at 02:06:41 UTC from IEEE Xplore. Restrictions apply.

only accounting for 3 of the 63 sensor bugs. Among these 3 bugs, only 1 exhibits serious symptoms. All sensor bugs triggered by healed sensor instances are timing sensitive. First, a sensor must fail, *and then* the sensor must recover later.

5) Interrupt Storms: Broken sensor hardware can trigger an *interrupt storm* where the hardware repeatedly triggers interrupts. During an interrupt storm, CPU time becomes dominated by interrupt handlers. This causes the firmware to starve, which can lead to UAS crashes. We find that 5 of 63 sensor bugs are triggered by interrupt storms. The firmware must ensure that interrupts are disabled during an interrupt storm. Unfortunately, this also keeps well-behaved devices that share a bus with the faulty hardware from communicating with the CPU. Redundant sensor instances should be placed on separate buses to increase UAS resilience to interrupt storms.

B. Simulating Sensor Bugs

Fault Injection is a technique where software is exposed to faults to determine if they are handled correctly. Testers undertake a *fault injection campaign* where faults are systematically injected to identify software defects. A fault injection campaign is driven by a *fault injection engine* that repeatedly runs the software under test and injects faults. The fault injection engine uses an *oracle* to judge the behavior of the software under test. The oracle typically works by monitoring a set of invariants that correct software must preserve. Whenever the oracle determines the software has misbehaved (e.g. an invariant is violated), the fault injection engine generates a bug report containing the details necessary to reproduce the scenario.

UAS firmware can be executed by a fault injection engine using software-in-the-loop (SITL) simulation. In a SITL simulation, the target firmware executes using a simulated vehicle. Figure 5 shows an overview of SITL in a fault injection framework. The HAL sends actuation commands to the simulator. The simulator models the effects of the actuation on the UAS's state. The new state is sent to sensor drivers in the HAL that simulate a sensor reading.

Sensor bugs triggered by complete sensor failures are good candidates for identification via fault injection for two reasons. First, developers struggle to identify these bugs in their code, especially ones with strong timing constraints. However, they represent a significant challenge for developers, accounting for nearly 40% of dangerous bugs. Second, this triggering condition can be simulated effectively. Both ArduPilot and PX4 support failing different sensor instances out of the box. For instance, ArduPilot supports failing the GPS and Barometer by configuring the "SIM_GPS_DISABLE" and "SIM_BARO_DISABLE" parameters respectively. For sensor types that do not have existing instrumentation to fail sensor instances, we create our own using each firmware's existing HAL APIs. The existing simulation sensor drivers in the HAL can communicate with a fault injection engine to fail sensor readings on demand.

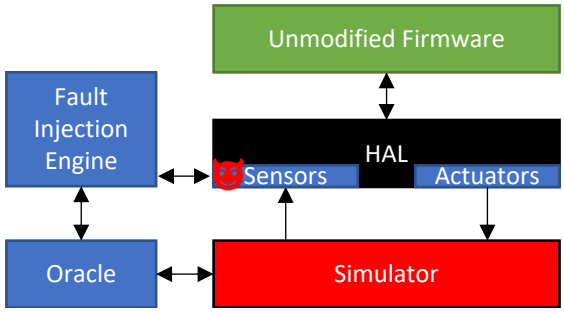

Fig. 5. Overview of fault injection setup.

There are two simple invariants that can detect dangerous bugs:

1) *Safety* - the UAS does not collide with an obstacle.
2) *Liveliness* - the UAS executes the pilot's commands, or executes a failsafe.

Safety can easily be tested by checking the simulation for a collision. Liveliness can be checked by comparing the UAS's physical state under a sensor failure and without a sensor failure.

VI. Testing Framework for Sensor Bugs

We reintroduced 5 previously reported sensor bugs in ArduPilot and PX4 using Avis [24] in order to better understand how sensor bugs affect UAS. We characterize their manifestations based on setting modifications, affected modes, sensors, time to symptom appearance, and additional requirements needed to trigger the bug. Here, we describe the bugs we reintroduced, discuss our reproduction methodology, and share our experiences.

A. Reintroduced Bugs

The bugs we reintroduced were: APM-4455, APM-4757, APM-5428, APM-9349, and PX4-13291. We selected these bugs because they are (1) sensor bugs, (2) triggered by sensor failures, and (3) share symptoms common with other bugs. Here, we describe each bug. Note that APM-9349 and PX4-13291 are described in Sections IV-A.2 and V-A.3 respectively.

1) APM-4455: APM-4455 is triggered when the UAS takes off into a GPS-aided flight mode, and then loses the GPS for a long period of time. Pilots expect the firmware to fallback to non GPS-aided position estimation. However, the firmware incorrectly continued to use stale GPS measurements. Over time, this bug caused position drift. This bug was patched by introducing logic to change the EKF's aiding mode after the GPS has timed out.

2) APM-4757: Firmware divides EKFs into multiple cores, each one using different sensor instances for redundancy. When a core becomes unhealthy (e.g. due to a failed sensor) the EKF switches its active core. Since different instances of sensors can have different biases due to their placement on the UAS, state estimates differ between EKF cores. The navigation subsystem must (1) detect when an

Authorized licensed use limited to: The Ohio State University. Downloaded on May 21,2023 at 02:06:41 UTC from IEEE Xplore. Restrictions apply.

TABLE III
SUMMARY OF EXPERIENCE REPRODUCING KNOWN BUGS USING FAULT INJECTION.

Bug ID	Default Settings?	Add. Modes?	Add. Failures?	Time until Detection	Add. Requirements?
APM-4455	✓	✗	✗	Long	✗
APM-4757	✓	✓	✓	Short	✓
APM-5428	✓	✓	✗	Long	✗
APM-9349	✓	✓	✗	Medium	✗
PX4-13291	✗	✗	✓	Short	✗

EKF core switches and (2) gracefully transition to using the new state estimate. In APM-4757, the EKF incorrectly reports the state innovation caused by EKF core switches. This causes the navigation subsystem to over-zealously command actuation, leading to unsafe maneuvers. Developers patched this bug by fixing the state estimation subsystem to accurately report innovation when the active EKF core switches.

3) APM-5428: UAS can safely operate without a GPS if they are controlled manually by a pilot. If the pilot later uploads an autopilot mission, firmware should refuse execution because it cannot estimate its absolute position without GPS. In APM-5428, the firmware incorrectly transitioned to autopilot when a mission was uploaded. This caused the vehicle's position to drift dangerously. This bug was patched by including a check that the GPS is healthy before a transition to a GPS-aided mode.

B. Methodology

To reintroduce each bug, we reverted the firmware back to the affected version. We launched the UAS firmware in SITL and executed a workload. Following the reproduction guidance in the issue report, we manually instructed a fault injector to fail the sensor instances needed to trigger the bug. We confirmed that each bug was detected using a simple oracle monitoring the invariants described in Section V-B. The remainder of this section describes each runtime feature we examined.

1) Settings: We checked if each bug affects additional flight modes beyond those listed in the incident report. To accomplish this, we ran several workloads using different manual and autopilot modes. We followed the same steps to reproduce the bug as described in the original incident report.

2) Additional Failures Triggers: Sometimes, the same bug can be triggered by failing sensors other than those listed in issues. We checked which of the bugs we reintroduced have this trait. We attempted to trigger each bug using sensor types other than the ones listed in the initial incident report.

3) Time Until Symptom Detection: We also examined how much time was needed for symptoms to be detected by the oracle after triggering each bug. We classify the time to symptom appearance as one of *Short*, *Medium*, or *Long*. Symptoms that appear shortly after the triggering condition take less than 3 seconds to manifest. We describe a medium time to appear as 3-9 seconds. A long manifestation time is ≥ 10 seconds.

4) Additional Triggering Requirements: Finally, we looked at what additional requirements are necessary to trigger each bug. Additional requirements refer to conditions beyond those explicitly stated in the incident report.

C. Results

Table III summarizes our experience reproducing existing bugs using fault injection. We find that default settings are sufficient to reproduce most bugs.

We were able to trigger 2 sensor bugs using sensors other than those described in the initial incident report. APM-4757 was initially triggered with a single compass failure. We were able to trigger this bug by failing a single IMU instance on the same EKF core. This caused an EKF core switch with the same symptoms as the original bug report. PX4-13291 could also be triggered with a compass loss. This caused the firmware to lose its local position estimate. Then, we triggered the battery failsafe. Without a local position estimate, the firmware failed to execute a failsafe. We verified that the patch for both bugs corrected the bug for other sensor types.

APM-4757 had additional conditions necessary for the bug to be triggered. Specifically, the EKF switch-over must introduce enough novelty to cause a position jump. However, we find that this condition can be met by waiting until late in the workload execution to trigger the bug. This works because the estimation errors due to sensor noise become worse over time. We also find that we can manipulate the readings from secondary sensor instances to cause the jump during EKF switch-over to destabilize the UAS and cause a crash. The merged bug patch prevents this behavior.

We found that several bugs could be triggered in modes not mentioned in their incident reports. For instance, APM-5428 can be triggered in any GPS-aided mode. The 2 bugs that did not fulfill this criteria used broad language to describe affected modes. We observe that if a mode has a sensor dependency on S then it will be affected by bugs triggered by S's failure.

Finally, we see that the time to manifest symptoms varies from bug to bug. However, for each bug, the time to manifestation was independent of the type of sensor failure that triggered the bug and the workload. Time to manifestation mostly depends on the type of symptom. For instance, APM-5428 manifests as position drift. It takes several seconds for the position to drift enough to be considered abnormal.

Authorized licensed use limited to: The Ohio State University. Downloaded on May 21,2023 at 02:06:41 UTC from IEEE Xplore. Restrictions apply.

D. Threats to Validity

Bug studies can present validity concerns [16], [27]. Potential threats to validity include our choice of firmware platforms, the bugs reported in the study and bias in our methodology.

We selected ArduPilot and PX4 because of their popularity, the size of their code bases, and their accessibility as open source projects. We believe they capture a wide range of features in UAS control firmware, but we also acknowledge that proprietary platforms (e.g., the DJI software development stack [7]) may differ substantially. Also, cutting edge platforms designed for emerging applications, e.g., SoftwarePilot [5] and AeroStack [21], may exhibit different patterns in future bug studies.

We strove to study a comprehensive sample of bugs across these platforms. We searched code repositories for key words related to patches, such as bug, problem, fix, etc. Our study excluded bugs that were not returned by our search criteria. We also manually analyzed each patch and developer comments to extract root causes, symptoms and other aspects of the bugs. We will release a spreadsheet with our selected bugs and their associated features. This allows researchers to replicate our results. Also, it reduces the burden on our methodology which focused on simple statistical measures. Researchers can explore more complex analysis, e.g., NOVA tests, to extract additional findings.

Overall, we do *not* believe our results should be extrapolated to all UAS control firmware platforms forever. However, we do believe our results capture the characteristics and runtime dynamics of bugs in modern, open-source, and widely used control firmware. Importantly, we report results that have intuitive explanations based on the function of UAS, providing confidence in the findings of our study.

VII. Discussion

Our bug study has implications for developers working on control firmware, researchers studying reliability and fault tolerance, and UAS users. For developers, our bug study can inform software engineering. For example, code base commits to sensor drivers should undergo rigorous unit tests, because mistakes are likely to cause crashes and other severe symptoms. Control firmware can construct unit tests based on simulation in realistic environments to test for bugs that produce symptoms only during flight. While simulations test some conditions that are not encountered in practice, our study shows that simulation can expand our understanding of bug symptoms. Finally, large code bases should clearly document the input, output, invariants and behavior of software components, reducing the frequency of semantic bugs where developers mix up supported features.

For researchers, the prevalence and impact of bugs in UAS control firmware present many opportunities. In-situ model checking uses simulation and/or real experiments to test execution invariants and has been used successfully to diagnose bugs in SSDs and filesystems caused by power outages [26], [13]. Custom static analysis processes (e.g. those in [9], [8]) have the potential to detect unit conversion bugs and incorrectly handled MAVLink messages.

Finally, our approach also impacts end users that may configure UAS under default settings not realizing that such settings could be triggering conditions in their environment. Using Rx approaches [20], [23], users can configure UAS to avoid these triggering conditions extending the lifetime of aircraft.

VIII. Conclusion

Reliability is paramount for UAS, especially as their use cases grow to include dangerous, remote, and sensitive missions. Bugs in UAS firmware can endanger nearby humans and jeopardize critical missions. However, how bugs affect UAS is not well understood. Engineering communities need data to build dependable UAS firmware. This paper examines 277 bugs affecting the ArduPilot and PX4 UAS firmware between 2016-2020. To the best of our knowledge, this is the first large-scale study of bugs in UAS firmware. We find that (1) **UAS bugs often have severe consequences** - 25% of bugs lead to vehicle crashes or other dangerous behaviors. (2) **Dangerous bugs are likely to impact users** - 80% do not have any special triggering conditions. (3) **Bugs in components handling sensor data contribute the most to dangerous behavior**, representing 30% of dangerous bugs. However, we find that (4) there is a light at the end of the tunnel: 43% of bugs can be reproduced in simulations, opening the door for automated bug-hunting techniques.

References

[1] Under the hood of ardupilot: Software quality and improvements. https://desosa.nl/projects/ardupilot/2020/03/26/under-the-hood-of-ardupilot-software-quality-and-improvements, 2020.

[2] ArduPilot. ArduPilot: Versatile, Trusted, Open. https://ardupilot.org, 2020. Accessed: 2020-05-23.

[3] Auterion. Auterion Enables Impossible Aerospace to Launch New US-1 Drone. https://auterion.com/auterion-enables-impossible_aerospace-to-launch-us-1-drone/, 2021. Accessed: 2021-02-26.

[4] Brendan Barry, Cormac Brick, Fergal Connor, David Donohoe, David Moloney, Richard Richmond, Martin O'Riordan, and Vasile Toma. Always-on vision processing unit for mobile applications. *IEEE Micro*, 35(2), 2015.

[5] Jayson G Boubin, Naveen TR Babu, Christopher Stewart, John Chumley, and Shiqi Zhang. Managing edge resources for fully autonomous aerial systems. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 74–87, 2019.

[6] H. Chen, W. Dou, Y. Jiang, and F. Qin. Understanding exception-related bugs in large-scale cloud systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 339–351, 2019.

[7] DJI. DJI Developer. https://developer.dji.com, 2021. Accessed: 2021-02-26.

[8] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, USA, 2000. USENIX Association.

[9] Dawson R. Engler. Incorporating application semantics and control into compilation. In *Conference on Domain-Specific Languages (DSL 97)*, Santa Barbara, CA, October 1997. USENIX Association.

[10] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Alfred Qi. A comprehensive study of autonomous vehicle bugs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 385–396, 2020.

Authorized licensed use limited to: The Ohio State University. Downloaded on May 21,2023 at 02:06:41 UTC from IEEE Xplore. Restrictions apply.

[11] Hu Huang, Samuel Z Guyer, and Jason H Rife. Detecting semantic bugs in autopilot software by classifying anomalous variables. *Journal of Aerospace Information Systems*, 17(4):204–213, 2020.

[12] S. Jha, S. Banerjee, T. Tsai, S. K. S. Hari, M. B. Sullivan, Z. T. Kalbarczyk, S. W. Keckler, and R. K. Iyer. Ml-based fault injection for autonomous vehicles: A case for bayesian fault injection. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 112–124, 2019.

[13] Yanyan Jiang, Haicheng Chen, Feng Qin, Chang Xu, Xiaoxing Ma, and Jian Lu. Crash consistency validation made easy. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 133–143, New York, NY, USA, 2016. Association for Computing Machinery.

[14] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47(6):77–88, 2012.

[15] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *ASID'06*, ASID'06: 1st Workshop on Architectural and System Support for Improving Software Dependability, pages 25–33, December 2006. ASID'06: 1st Workshop on Architectural and System Support for Improving Software Dependability ; Conference date: 21-10-2006 Through 21-10-2006.

[16] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 329–339, 2008.

[17] MAVLink. MAVLink Developer Guide. `https://mavlink.io/en/`, 2020. Accessed: 2021-02-26.

[18] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[19] PX4. PX4 Autopilot: Open Souce Autopilot for Drones. `https://px4.io`, 2020. Accessed: 2020-05-23.

[20] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 235–248, New York, NY, USA, 2005. Association for Computing Machinery.

[21] Jose Luis Sanchez-Lopez, Ramón A Suárez Fernández, Hriday Bavle, Carlos Sampedro, Martin Molina, Jesus Pestana, and Pascual Campoy. Aerostack: An architecture and open-source software framework for aerial robotics. In *International Conference on Unmanned Aircraft Systems*, 2016.

[22] Jose Luis Sanchez-Lopez, Martin Molina, Hriday Bavle, Carlos Sampedro, Ramón A Suárez Fernández, and Pascual Campoy. A multi-layered component-based approach for the development of aerial robotic systems: the aerostack framework. *Journal of Intelligent & Robotic Systems*, 88, 2017.

[23] Christopher Stewart, Kai Shen, Arun Iyengar, and Jian Yin. Entomomodel: Understanding and avoiding performance anomaly manifestations (winner of best paper award). In *IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2010.

[24] Max Taylor, Haicheng Chen, Feng Qin, and Christopher Stewart. Avis: In-Situ Model Checking for Unmanned Aerial Vehicles. In *The 51st annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021.

[25] Christopher Steven Timperley, Afsoon Afzal, Deborah S Katz, Jam Marcos Hernandez, and Claire Le Goues. Crashing simulated planes is cheap: Can simulation detect robotics bugs early? In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 331–342. IEEE, 2018.

[26] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. Torturing databases for fun and profit. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 449–464, USA, 2014. USENIX Association.

[27] Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 913–923, 2015.

Authorized licensed use limited to: The Ohio State University. Downloaded on May 21,2023 at 02:06:41 UTC from IEEE Xplore. Restrictions apply.