Detecting Semantic Bugs in Autopilot Software by Classifying Anomalous Variables

Hu Huang *, Samuel Z. Guyer †, and Jason H. Rife ‡
Tufts University, Medford, MA, 02155

Like any software, manned-aircraft flight management systems (FMSs) and unmanned aerial system (UAS) autopilots contain bugs. A large portion of bugs in autopilots are semantic bugs, where the autopilot does not behave according to the expectations of the programmer. We construct a bug detector to detect semantic bugs for autopilot software. We hypothesize that semantic bugs can be detected by monitoring a set of relevant variables internal to the autopilot. We formulate the problem of identifying these variables as an optimization problem aimed at minimizing the overhead for online bug detection. However, since the optimization problem is computationally prohibitive to solve directly, we utilize graph-based software models to identify a suboptimal solution. In analyzing real and injected bugs within a particular block of code (a program slice), our proof-of-concept approach resulted in a model using only 20% of the variables in the slice to detect real and synthetic bugs with a specificity of 99% and a sensitivity of at least 60% for all bugs tested (and 90% or higher for many of them).

Nomenclature

P_{system}	=	failure probability of the entire system due to a hidden bug.
$P_{pre\text{-}service}$	=	failure probability of pre-service testing.
$P_{except,md}$	=	failure probability of the exceptions handler.
$P_{detector,md}$	=	failure probability of the bug detector.
V	=	the set of all the variables in the autopilot software.
P	=	$P \subseteq V$, the set of the variables chosen for monitoring in order to detect software bugs.

I. Introduction

Even though modern autopilots can complete complex tasks [1], they often suffer the same ailments as other modern software: software bugs. Software bugs can also be called "defects" or "faults," and these terms will be used

*PhD student, Dept. of Computer Science, 161 College Ave, Non-Member
†Associate Professor, Dept. of Computer Science, 161 College Ave, Non-Member
‡Associate Professor, Dept. of Mechanical Engineering, 200 College Ave, Non-Member

interchangeably in this paper [2]. Defects in autopilots have obvious real-world consequences. For autopilots in small drones, the most severe consequence of a software defect is when the drone suddenly falls out of the sky or crashes. A crash may destroy the drone, and more importantly, may cause loss of life or substantial damage to property. While most drones that fall out of the sky are harmless [3], invariably some have fallen on people [4]. For commercial aircraft, even though software defects occur infrequently, there are still documented instances. These defects include possibly causing an engine shutdown [5], causing the Flight Management System (FMS) to make a wrong turn [6], and forcing down the nose of the aircraft when the autopilot incorrectly sensed that the plane was stalling, which tragically resulted in two fatal crashes [7, 8]. Although our focus in this paper is aircraft, it is notable that software defects more broadly affect Cyber-Physical Systems (CPS), including self-driving cars [9–11].

To help provide robustness to latent bugs and possibly also to streamline aviation software verification, we envision an online bug detection system that would continually check for software anomalies. This paper demonstrates the concept through implementation and testing of a prototype bug-detection system and evaluation using an open source autopilot called Ardupilot. Our contributions are as follows:

1) Established the feasibility of bug detection using a snapshot sample of internal variables. The approach detected real bugs in an open-source flight-control code.
2) Demonstrated effectiveness of program slicing (and specifically of backward slicing) as a basis for improving bug detector performance, by providing more sensitivity in a local region of code. This demonstration was conducted by applying our bug detection tool to Ardupilot and assessing performance for real and injected bugs.

The rest of this paper is organized as follows. Section II discusses related research, Section III describes the bug detector implementation and the bugs that we will use in our experimental evaluation. Section IV frames our variable selection problem as an optimization problem, which we solve heuristically using methods detailed in Section V. We assess the performance of our approach via experimental evaluation as described in Section VI. Results are presented in Section VII and discussed in Section VIII. A brief summary concludes the paper.

II. Background

To mitigate against software bugs, various solution approaches have been formulated including programming best practices, testing, formal methods and bug detection. We discuss each below.

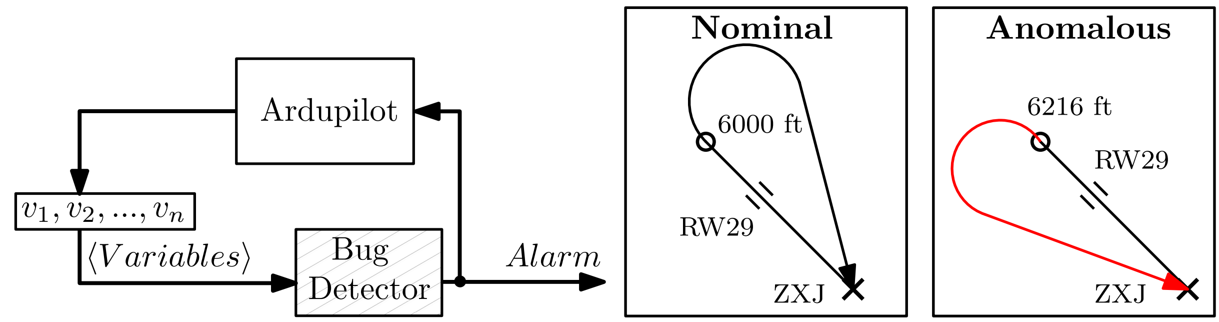
Programming best practices remains a popular approach to prevent software bugs and ensure software quality. Software engineering teams often follow best practice standards when designing for CPS, such as MISRA C [12] and CERT [13]. These standards restrict the use of "unsafe" programming language features in order to make the software more reliable. Some examples of "unsafe" language features include use of `goto`'s, dynamic memory allocation beyond program initialization, and use of `union`'s.

A second approach to mitigate against bugs is through testing. Although testing is considered an industry best-practice, as documented in RTCA DO-178C [14], it is also costly. Testing can reveal hidden bugs, but nevertheless it cannot guarantee that the software is bug-free.

A third approach uses formal verification, which *proves* the software to be correct by construction [15, 16]. Formal methods have recently been applied to constructing secure air vehicle software [17]. Even though formal verification is becoming more mainstream, this approach still faces two large barriers to wide-spread adoption. Firstly, formal verification techniques depend on a set of formal specifications written in temporal logics, which are difficult to understand and maintain for non-experts. Secondly, formal verification is not fully automated, making it difficult to scale for large programs. Due to these challenges, formal verification has remained an academic research topic that has not yet been integrated into commercial products.

A fourth approach is online bug detection. Online bug-detection tools have been developed in other application domains [18–22] but few tools have been developed for autopilot systems. Of the bug detection tools that have been proposed, they have generally focused on inputs and outputs of the autopilot software [23, 24]. No prior bug-detection efforts have focused on the internal variables of an autopilot; however, it is noteworthy that internal-variable scanning has been considered in analyzing the cybersecurity of an autopilot [25].

In this paper, we consider the fourth approach and pursue a practical bug monitoring implementation that leverages observations of internal variables to enhance performance. Our bug detector takes snapshots of the variables within the autopilot and utilizes Machine Learning (ML) models to evaluate each snapshot: to decide whether variables indicate a faulty program state. Fig. 1(a) shows a block diagram of our bug detector in relation to the autopilot. As shown in the figure, the bug detector continuously scans internal variables of the autopilot and issues an alarm flag to indicate when an anomaly is detected. Our bug detector aims to reduce the amount of required *pre-service verification* (also known as testing) and avoid requiring formal verification of the full autopilot.



(a) Feedback loop between bug detector and Ardupilot **(b) Rockwell Collins FMS Bug**

Fig. 1 Feedback loop 1(a) and a recently corrected semantic bug 1(b), illustration adapted from [6].

This paper focuses on the detection of semantic bugs in autopilots. Semantic bugs are errors in a program that do not cause the program to stop execution, but that cause values to be computed in an incorrect way and that result in a deviation of the program's behavior from the programmer's expectation. Semantic bugs are problematic because they are much harder to detect than exceptions that cause a program to stop running (or colloquially to *crash*). We focus on semantic bugs because they remain a major root cause of reported bugs even as software matures, accounting for over 70% of the bugs in three large open source projects [26]. This stands in contrast with memory bugs that decrease in number with software maturity [26, 27]. Fig. 1(b) illustrates an example of a semantic bug recently found in the Rockwell Collins FMS [6]. The nominal trajectory begins with take-off from ZKJ and an ascent to 6000 feet, then proceeds to a right-hand turn to return to a waypoint at ZXJ. However, if an operator updates the altitude at the waypoint, which is shown on the right pane, then the FMS silently *overrides* the operator and commands the aircraft to perform a left-hand turn to go back to ZXJ (highlighted in red). Changing the turning direction of the aircraft is highly dangerous and may lead the aircraft into oncoming traffic.

A. Other Related Work

Model-Based Fault Detection. In model-based fault detection [28], a physical model of the aircraft is first constructed from kinematic and dynamic equations. The model can then predict the state of the aircraft and compare the predictions against actual sensor values to obtain residuals. Residuals are small when there's no fault but will increase above a certain threshold when a fault appears. Model-based fault detection techniques assume that faults occur at sensors and actuators and do not address faults which arise in software.

Anomaly Detection in UAVs. Anomaly detection methods using ML models rather than physical models have also been applied to UAVs; however, the focus has been mostly on hardware faults and not software. Khalastichi et al. [23, 24] used supervised and unsupervised ML models online to detect injected faults in sensors.

Anomaly Detection in Desktop Software. There are two major approaches to anomaly detection in desktop software, value-based and invariant-based. Both approaches require a data collection phase where various locations in a program are instrumented and information gathered when the program is executed. In the value-based approach, a ML model learns to detect faults from values collected [19, 20]. In the invariant-based approach, no learning takes place. Instead, invariants are rejected when they are violated in subsequent program executions [18, 21]. Both approaches make heavy use of heuristics in the selection of variables. Chen et al. [22] is the closest work compared to our approach, instead of variables they attempted to select the optimal number of invariants to satisfy cost and coverage constraints. Our approach is complementary and examines a similar problem in variable selection but using more flexible models as compared to invariant-based approaches, which we believe will enhance monitor performance as required for aviation applications.

III. Methodology

The main focus of this paper is to implement a bug detector for flight software and demonstrate its ability to detect actual bugs from a development database. To this end, we have chosen an autopilot system that is widely used, reliable and open source. Namely, we are using the Ardupilot software system. In order to construct the bug detector it is necessary to instrument Ardupilot, both to collect data for training our model and subsequently to implement the online bug detector. This section discusses key elements of our implementation, including the Ardupilot software, our instrumentation approach, and our bug detector design.

A. Ardupilot

Ardupilot [29] lies at the heart of over one million drones [30] and stands out from other publicly available autopilots [31–33] as the most mature open-source codebase designed for unmanned vehicles. Ardupilot has even been used as the backbone on drone projects from large companies such as Microsoft [34] and Boeing [35, 36]. With well over half a million lines of C++ code, Ardupilot supports many features that include Software-In-The-Loop (SITL) testing, complex control algorithms, automatic takeoff and landing, and sophisticated mission planning. These advanced features make Ardupilot an ideal candidate for a bug detection implementation.

B. Data Collection From Ardupilot

In order to the collect data from Ardupilot, we have written an instrumentation library called Oscilloscope (OScope), which must be compiled together with Ardupilot. We modified the Ardupilot compilation process by using the Lower-Level Virtual Machine (LLVM) compiler framework [37] (version 4.0.0) to combine Ardupilot and OScope, as depicted in Fig. 2. Tools that we have created, such as the `slicer` and the `instrumenter`, are shaded in blue. One of the main advantages of using LLVM is the Intermediate Representation (IR), which can be manipulated independent of the source files. We harness this independence so that OScope can be compiled together with Ardupilot without requiring any modification to the original Ardupilot source code.

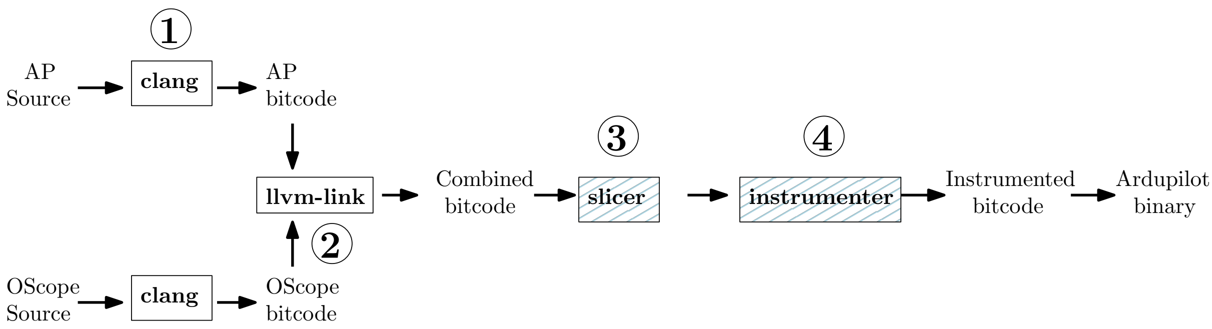


Fig. 2 Our compilation pipeline shows how we create an instrumented version of Ardupilot.

At the start of the compilation process, at step (1) in Fig. 2, we first leverage `clang` to lower both Ardupilot and OScope from the source code to LLVM bitcode, which is a binary format for the LLVM IR. Next, these two separate bitcode files are combined into one bitcode file using `llvm-link` at step (2). At step (3), the `slicer` identifies sites in the bitcode which are appropriate for instrumentation. Then the `slicer` passes those locations to the `instrumenter`. At step (4), the `instrumenter` adds calls to the OScope instrumentation API at the instrumentation sites and generates the instrumented bitcode. In the final step, once all the modifications are complete, the LLVM bitcode is transformed into x64 assembly and `gcc` compiles the x64 assembly down to binary, thereby creating an instrumented version of the Ardupilot binary. When this modified Ardupilot binary executes, OScope records the relevant values of variables that are being monitored and writes these values out to a log file, which is then analyzed off-line. More details on our tool can be found in the Supplemental Materials.

C. Bug Detector

The goal of the bug detector is to analyze variables in the primary autopilot in order to detect when a software anomaly (i.e. semantic bug) occurs. In order to detect semantic bugs, a bug detector requires two components: 1) a pertinent source of data and 2) a method to interpret and distinguish nominal from faulty data. In regards to the first component, which is the data source, we hypothesize that the values of variables *within* the autopilot serve as an information-rich data source to detect bugs. Previous work to detect semantic bugs has used a range of different data sources. Bug detectors focused on desktop software have used various data sources such as control flow [20], heap characteristics [38] and discovered invariants [18, 21, 22] with varying success. For bug detectors on autopilots, data sources such as sensor and actuator values [23, 24] or function call data [25] have been used. While the inputs and outputs of the autopilot software (sensor readings and actuator commands) are important data sources, they do not contain enough information to detect difficult semantic bugs. Indeed, in this paper we show experimentally several semantic bugs that are *essentially undetectable* if we only monitor the inputs and outputs. In this paper we demonstrate that variables *within* the autopilot will enable our bug detector to detect semantic bugs that are otherwise undetectable.

The second component that a bug detector requires is a method to distinguish correct versus faulty patterns in the data. We hypothesize that statistical methods are a set of powerful tools to accomplish this task. Previous work in bug detection have used logistic regression [39], hypothesis testing [40], conditional probability [41], clustering [42] and Markov Models [20]. While a wide variety of statistical ML models are suitable for bug detection, many models are opaque and cannot provide explanatory power to a human analyst. We believe that interpretable models, such as tree-based models, can provide much-needed insight into the decision process in comparison to opaque models such as neural nets. We also believe that current off-the-shelf models are sufficient to provide evidence to a human analyst that the system is faulty or likely going to be faulty. Hence we have chosen *not* to develop new statistical methods but to leverage existing ML models. Specifically, we chose Decision Tree and AdaBoost as the ML models for our bug detector as both algorithms can produce a set of decision rules that can be verified by a subject matter expert.

As another design decision, we chose an implementation that would isolate software anomalies from hardware faults, such as faults involving servos, sensors, or other electrical or mechanical components of the aircraft. Our solution to isolating software anomalies is to use a snapshot detector, where all variables are assigned essentially instantaneously.

Although focusing on snapshot data limits performance (as compared to say, a batch or sequential analysis comparing data over multiple time steps), we can say with confidence that the dynamics of the physical hardware system play no role in the monitoring, a key detail that differentiates software bugs from hardware faults, and which alleviates the need for machine learning to model physical dynamics in addition to software computations. In our instrumentation, we define snapshots to align with iterations of the main code loop, each lasting approximately 0.01 seconds. As Ardupilot executes, the bug detector continuously records data from each variable and appends this data to a log file, thereby creating one snapshot after another contiguously.

D. Real & Injected Ardupilot Bugs

One of the fundamental steps in building our bug detector is to gather data from simulated flights to train our ML models. These simulated flights included a mix between normal behavior and buggy behavior. In our work with Ardupilot, we have explored three semantic bugs identified in the Ardupilot Bug Database, including Bug 2835 (impulsive pitch change), Bug 6637 (faulty holding pattern), and Bug 7062 (failure to transition between flight modes). These bugs are described in more detail in [43]. These bugs all occur in different sections and releases of the code. Thus, for testing in this paper, we focused on a single one of these bugs, Bug 7062.

In addition to studying a real bug, we also considered a number of injected synthetic bugs. By injecting synthetic bugs, we can study a greater diversity of possible events while focusing on instrumenting one compact section (and one release) of the Ardupilot code. To ensure that our synthetic bugs are representative of bugs encountered "in-the-wild" and not merely small mutations of the source code [44], we injected bugs that cause faulty branching, much like Bug 7062 and other decision-logic bugs identified in the wider literature [45, 46]. Specifically, we injected bugs that, when active, forced the program to execute only certain branches of an `if` statement. In all we considered nine such injected bugs for the purposes of training and testing our ML models. More information on Bug 7062 and our injected bugs is provided in the Supplementary Materials, posted online with the article.

IV. Quantifying the Bug-Detector Design Problem

Our bug detection technique depends on selecting a set of variables for monitoring. The problem of selecting this set of variables is nontrivial because of the need to balance design requirements, which include: overhead, sensitivity, specificity, coverage and alert-time. These design requirements are all interdependent in various ways.

Overhead is defined as the additional processing and memory required to run the bug-detection algorithm alongside the original program. Overhead is difficult to predict precisely at the design stage, so it is useful to introduce a metric which is easier to quantify as a surrogate for overhead. In this paper, we approximate overhead as the number of variables monitored n. As the size of the set of variables increases, the overhead from monitoring will also increase monotonically and degrade the overall performance of the autopilot. Therefore, it is desirable to limit the number of variables being monitored.

However, there is a trade-off. If the number of variables being monitored is too low, the bug detector may be losing crucial information to perform its function. Accordingly, bug detection performance must also be quantified in terms of other criteria, including:

- *Specificity*: The probability that the bug detector withholds an alarm when bugs are absent.
- *Sensitivity*: The probability that the bug detector alarms within a specified alert time after a hazardous bug occurs.
- *Coverage*: The number of lines in the code for which the bug detector provides a target level of sensitivity.
- *Alert-time*: The allowable time between the onset of a bug and the moment it begins to threaten system safety.

We expect certification agencies to set safety standards on these design criteria such that there are upper limits on allowed alert-time and lower limits on coverage, sensitivity and specificity. However, there is one design criterion which is unconstrained: overhead. This is largely a matter of how much processor and memory we are willing to throw at the problem. As such, there is an opportunity to try to minimize overhead to keep processor and memory costs as low as possible (given that we can meet our design criteria). In short the design problem can be framed as an optimization problem, where the overhead is the cost function that must be minimized. The other specifications are constraints.

$$\begin{aligned} \min_{P \subseteq V} \quad & \text{overhead}(P) \\ \text{subject to} \quad & \text{sensitivity}(P) \geq b_1, \\ & \text{specificity}(P) \geq b_2, \\ & \text{coverage}(P) \geq b_3, \\ & \text{alert-time}(P) \leq b_4 \end{aligned} \tag{1}$$

The decision variables in the above optimization problem are the set of variables P which are fed to the machine-learned classifier in real time. The variables in P are a subset of all the variables V in the software ($P \subseteq V$). As mentioned above, overhead is approximated as the cardinality of the set P, since we expect the processor and memory costs to shrink monotonically with the number of variables being monitored. Like the objective function, the constraints defined above also depend strongly on the set of variables P used to train and implement ML models in the bug detector.

It is not the goal of this paper to find the global minimum of Eq. (1). Rather, the optimization problem was framed to provide a clear statement of our bug-detector design problem. As it turns out, Eq. (1) is difficult to solve directly, in large part because most of the criteria are difficult to evaluate and because the evaluation is computationally prohibitive. For example, in Ardupilot the set V consists of approximately 15,000 variables in Ardupilot, so the design space is the power set of V, $\mathcal{P}(V)$. Instead of seeking a global optimization solution, it is more practical to form heuristics that attempt to satisfy constraints while reducing overhead as much as possible.

Our approach in this paper will be to find a heuristic, suboptimal solution to a slightly relaxed form of the full design problem in Eq. (1). Our relaxation will simplify the problem by reducing the required coverage to one small section of code rather than the full code. Moreover, we will set specific values for sensitivity and specificity targets (0.7 and 0.99 respectively), using supervised ML models and labeling nominal and buggy cases. Lastly, we note that our snapshot implementation provides instantaneous detection, so the alert-time is simply treated as being one time step. The following relaxed design problem results. Note that the *min* operator is replaced with *reduce* to indicate that we seek to improve overhead, and that we can be satisfied even if the global minimum is not found.

$$\begin{aligned} \underset{P \subseteq V}{\text{reduce}} \quad & \text{overhead}(P) \\ \text{subject to} \quad & \text{sensitivity}(P) \geq 0.7, \\ & \text{specificity}(P) \geq 0.99 \\ & \text{alert-time}(P) \leq 0.01\ s \end{aligned} \tag{2}$$

In evaluating our monitor designs against the above constraints, alert-time is automatically satisfied by monitor construction. Specificity and sensitivity are evaluated heuristically. Specificity is assessed using a finite number of training data points to compute the number of true negatives (TN) normalized by the total number of true negatives and false positives (FP) for each variable set P, such that: $\text{specificity}(P) = \frac{\text{TN}(P)}{\text{TN}(P)+\text{FP}(P)}$. Similarly, sensitivity is assessed used a finite-size testing data set to compute the total number of true positives (TP) normalized by the number of true positives and false negatives (FN) for each variable set P, such that: $\text{sensitivity}(P) = \frac{\text{TP}(P)}{\text{TP}(P)+\text{FN}(P)}$.

The specificity and sensitivity criteria have been set leniently to provide a baseline assessment of feasibility. Specificity is the primary driver, because a low specificity implies a large number of false alarms, which would make the system unusable. As such we set our feasibility target for specificity to 0.99. Sensitivity is important only in that high sensitivity enables a reduction of pre-service verification requirements, as will be discussed later in this paper. Since bug detectors are not currently deployed in aviation systems, any sensitivity better than zero is useful. As such, we set our feasibility target for sensitivity to be a modest value of 0.7. Importantly, if we can achieve this modest baseline, then it will be worth considering more advanced methods in the future (extending analysis over multiple time steps, employing more advanced ML methods, and introducing optimized variable-set selection to achieve higher levels of specificity and sensitivity as desired for aviation implementation [47]).

V. Solution Approach

With the goal of addressing our relaxed design problem, we introduce two heuristic strategies to select the set of variables P that the bug detector will use for detection. The two heuristic approaches for defining probe sets include using (i) system inputs and outputs and using (ii) variables extracted locally from a program slice. Variables from the system inputs and outputs are abbreviated as SysIO. SysIO variables are determined by examining the source code and finding the variables that interfaced with external hardware (e.g. sensors or actuators). In the second set of variables, we exploit the underlying structure of the autopilot software by considering only variables local to the bug. Local lines of code and their associated variables are extracted using a process called program slicing, which is a well established technique in the programming languages community [48–50]. The goal of computing the program slice is to select variables that are on a shared data-flow path. Presumably, if the bug lies in the program slice, variables on a shared data-flow path are particularly sensitive to the bug. We elaborate on both variable sets below.

A. SysIO Variables

We identified 28 variables which are the input and output channel variables that are sent to Ardupilot via radio control (RC). Of the 28 variables chosen, 14 are RC inputs and 14 are RC outputs. For example, RC input channels 1-4 map, by convention, to Roll, Pitch, Throttle, and Yaw commands.

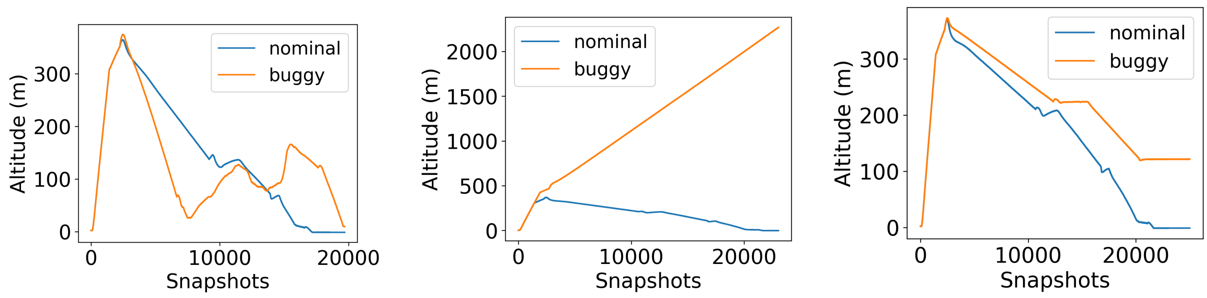
B. Program Slice Variables

In order to evaluate our hypothesis that local variables (identified by program slicing) improve detection performance, we ran ML algorithms to process all of the variables in the entire slice. This variable set is labeled SliceFull. Because the number of slice variables is relatively large (hundreds of variables), we also considered additional heuristics to prune the set. Specifically, two subsets of slice variables were considered: the inputs and outputs of the program slice (SliceIO) and the nodes in the dominance frontier within the slice (SliceDF). The Dominance Frontier can be determined in linear time using an algorithm from Cooper et al. [51]. The logic of selecting these subsets is that the variables in the slice are expected to be highly correlated (because these variables share the same data-flow path), so it is likely that many of the variables in the slice provide redundant information and therefore increase overhead without substantially increasing bug detection performance.

The specific section of Ardupilot from which we have generated our program slice correspond to the Total Energy Control Systems (TECS) module. Importantly, this module contains an adequate number of conditional statements that can be appropriated for synthetic bug injection. The module also contains a real semantic bug from the Ardupilot bug database: Bug 7062 (which is described in more detail in the Supplemental Material). Note that the purpose of the TECS module is to maximize climb performance and aircraft endurance by modulating pitch angle and throttle to control height and airspeed.

VI. Experimental Evaluation

We perform a series of experiments and gather data from Ardupilot in order to assess the variable selection approaches discussed in Section V. We follow the experimental protocol below in our experiments: **(A)** Introduce bugs into the Ardupilot source code, one at a time from the list of nine synthetic bugs and one real bug labeled Bug 7062 in



(a) Early Descent **(b) Uncontrolled Climb** **(c) Waypoint Overshoot**

Fig. 3 Examples of consequences of injecting synthetic bugs.

the Ardupilot Bug Database, **(B)** Simulate flights, **(C)** Store instrumented data to disk for four probe sets, including SysIO, SliceFull, SliceIO, and SliceDF, **(D)** Process data, **(E)** Train models on half of stored data, **(F)** Evaluate models on remaining stored data. We now discuss each step in more detail below.

A. Introduce Bugs

As discussed above, we focused our synthetic bug injection efforts on the TECS module in Ardupilot. Each synthetic bug, labeled with an integer identifier 0 through 8, corrupts an existing `if` statement, forcing the program always to execute only one particular branch.

For the purposes of training and testing, we simulated Ardupilot as applied to a quadplane, a specialized unmanned aircraft that is configured with quadrotors fixed to the aircraft in front and behind each wing, to provide a VTOL capability for an otherwise conventional fixed-wing aircraft. The real bug (Bug 7062) involves an occasional failure to transition from vertical takeoff to level flight. The nine synthetic bugs exhibited different consequences, falling into one of the following three categories:

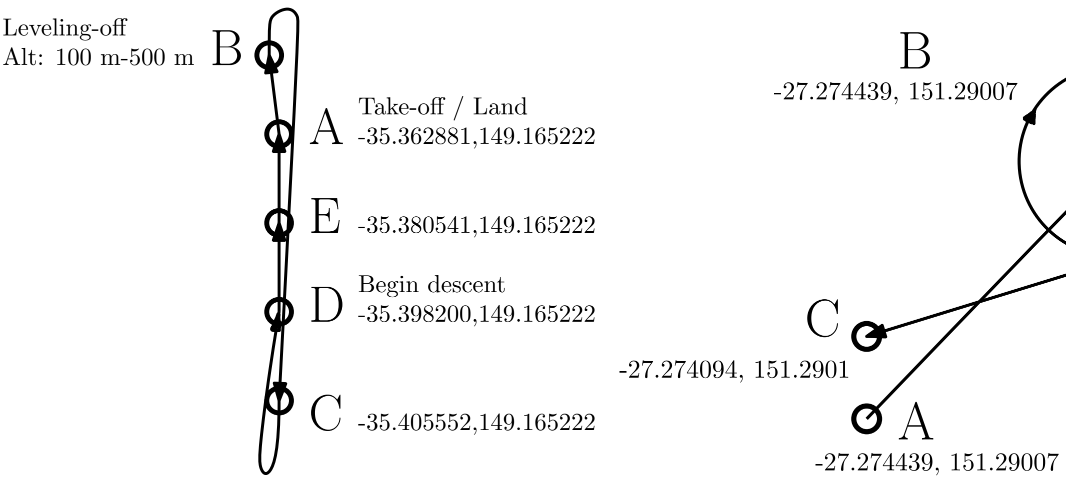
- **Early Descent** - This consequence is exhibited by 3 out of 9 bugs (bugID: 0,3,4). As shown in Fig. 3(a), these injected bugs cause the plane to descend earlier and faster relative to a nominal flight. Sometimes the plane corrects itself, pulls out of the descent, and climbs back to a safe altitude; however, the plane sometimes fails to climb and crashes into the ground.
- **Uncontrolled Climb** - This consequence is exhibited by 2 out of 9 bugs (bugID: 1,2). In these cases as shown in Fig. 3(b), the plane attempts to navigate each waypoint but does not keep to the commanded altitude. Instead the plane steadily climbs throughout the entire flight, which eventually causes the simulation to time out, because the plane never lands.
- **Waypoint Overshoot** - This consequence is exhibited by 4 out of 9 bugs (bugID: 5,6,7,8). As depicted in Fig. 3(c), the plane flies normally for most of the duration of the flight, until it reaches the last waypoint. However, once the plane gets close to the last waypoint it does not descend but instead continues past the waypoint. This is

the most deceptive of the injected defects because the flight is normal up until the onset of the incorrect behavior, which only appears near landing.

B. Simulate Flights

We simulate all flights using Ardupilot with JSBSim, an open-source flight dynamics engine [52]. All flights are executed on a server with 12-core 2.8 GHz Intel Xeon Processors, 12 Gigabytes of RAM and running Archlinux 4.8.13. Data for training and testing was collected by running 20 trials for each bug and 20 trials for nominal conditions. Simulations typically running for 5-8 minutes, with the simulation terminating after the aircraft completes its landing. Time steps in which the consequences of the bug manifested were classified manually to enable supervised learning.

For the evaluation on injected bugs, the flights are based on the SIG Rascal 110 RC fix-wing plane, which has a wingspan of 9.17 ft, a wing area of 10.57 ft and a flying weight of 13 lbs. We based each trial on a flight plan with five waypoints, an example is shown in Fig. 4(a), which shows the waypoints and their coordinates. The Rascal followed the path from $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow A$. Waypoint A was fixed at $-35.362\,881°$ latitude, $149.165\,222°$ longitude for all trials. We varied the latitudes for the other waypoints and fixed the longitudes. Each trial was generated by first selecting a travel distance uniformly chosen between 1.25 km and 5 km, either north or south of waypoint A. Waypoint B and C were the same altitudes, which varied uniformly between 100 and 500 meters inclusive. The latitude for waypoint C was perturbed an additional amount chosen uniformly between $-0.002°$ and $0.002°$. The random horizontal travel distance and a second random parameter, a glide slope uniformly selected between $1°$ and $5°$ constrained the altitude of waypoint D, where the plane must begin descent. Waypoint E was located in the middle of the descent, where we specified a 10% chance the plane would not land but perform a go-around.



(a) Scenario for Injected Bugs **(b) Scenario for Real Bug**

Fig. 4 Flight maps for injected bugs and the real bug.

Although severe consequences, as described by Fig. 3, occasionally manifested for all synthetic bugs in the context of the above flight plan, a second flight plan was needed to trigger the real bug, which only appears during a transition between VTOL and level flight, a condition not present in our baseline testing scenario. For tests involving the real bug, we used the a second flight plan as depicted in Fig. 4(b). The second flight plan consisted of three waypoints. Our model aircraft was again a SIG Rascal, but with one addition: a set of four rotors attached to the airframe in the vertical direction to enable VTOL flight. The plane took-off from waypoint A towards waypoint B, where it reached an altitude uniformly chosen between 100 meters to 300 meters. Once the plane had transitioned to fixed-wing flight, it then circled waypoint B four times, performing a "loiter" manuever. Each simulation trial had a 50% chance to trigger Bug 7062. If the trial was to trigger Bug 7062, the loiter manuever was performed below the altitude at waypoint B, which was chosen uniformly between 30 meters and 20 meters below the altitude of waypoint B. If the trial should not trigger Bug 7062, then the loiter was performed at an altitude chosen uniformly between 20 to 100 meters above the altitude at waypoint B. Once the loitering maneuver was finished, the plane traveled to waypoint C and landed.

For both the baseline flight plan and the VTOL flight plan, we added further variation by changing the wind in each trial. Wind was controlled using three different parameters: wind direction (degrees), wind speed (m/s) and turbulence (m/s). Wind direction was chosen uniformly between 0 and 359 degrees inclusive, wind speed was chosen from a normal distribution with mean of 6.5 m/s and deviation of 2. The simulation turbulence model was configured with two parameters, including a mean variation of 0.5 m/s and a standard deviation of 1 m/s.

C. Store Instrumented Data

The OScope utility (see Fig. 2) stored a batch of variable values once per iteration of the main loop of Ardupilot, which amounts to a sample rate of approximately 100 Hz. Note that we ran the simulation in real time because the run-time behaviors of the autopilot software were not representative when accelerating the simulator to run in faster-than-real-time. Data records were flexible in structure because some variables were updated more than once per time step (in which case all updated values were stored) and because some variables were occasionally not updated during a time step (in which case no variable values were stored). For the purposes of regularizing the data, only one value of each variable was used for training. Specifically, the last value from each time step (in the case of multiple updates) or the last updated value (in the case of no update).

Table 1 Number of variables in each of the four variable sets evaluated

Variable Set	Set Size
SliceFull	283
SliceIO	84
SliceDF	56
SysIO	28

For each simulation trial, variables were stored for all time steps for four different probe sets. The number of variables in each variable set is specified in Table 1. The smallest of the variable sets is SysIO, followed by SliceDF, SliceIO, and SliceFull. Note that all of the SliceDF and SliceIO variables are contained with in the SliceFull set. There is an overlap of 31 variables between SliceDF and SliceIO. There is no overlap between SysIO and any of the other variable sets.

D. Process Data

After acquisition, data were formatted and balanced to support training and testing of ML algorithms. Half the data were used for training and the other half were reserved for testing. For each trial, we manually examine the data to classify time steps where the bug manifested. In this way, we created labels to support supervised learning with two classes: nominal and buggy. We understand the that our approach is limited, and our intent is to pursue one-class classification (in which training data are assumed to be nominal) in future work, as described in [47].

To be precise, 10 nominal and 20 buggy trials were recorded for each of the bugs. 10 nominal and 10 buggy trials were used to train the ML algorithms and 10 buggy trials were reserved for testing. Given that 10 bugs were analyzed, the total number of trials considered is 300. Each synthetic-bug trial resulted in 25,000 to 35,000 snapshots per trial (i.e. a data record of 250-350 s in duration). The real bug case resulted in 80,000 to 140,000 snapshots per trial (i.e a data record of 800 to 1400 s).

A key step to balancing the data was to match the length of data sets used to train each ML classifier. For this purpose, the data set for each bug was associated with a nominal data set, and the longer data record was shortened so the duration of the two data records matched. The purpose of this balancing was to ensure a similar number of data points were available for buggy and nominal runs. Balancing data record duration was particularly relevant for cases in which there was a crash (shortening buggy data record) or an uncontrolled climb (lengthening buggy data record). Furthermore, because bugs, when present, were only active for a portion of each trial, data were checked to confirm that at least 3000 snapshots (approximately 30 seconds of data) were classified as bug-active cases.

Data were processed to provide variable values and *deltas*, which were defined to be the differences between each variable and its most recent prior value. Both the raw values and the deltas were made available for training and testing.

E. Train Models

Since this paper is intended to be a feasibility study, a different classifier was defined for each bug in order to capture a reasonably good level of performance, as might be expected for a well-chosen classification surface. For each of the ten bugs (nine injected and one real) we trained both Decision Tree and AdaBoost models. We used the default parameters as specified in the `scikit-learn` library, a standard Python library supporting ML applications. Training was conducted to achieve a specificity of 0.99, a specification consistent with (2).

F. Evaluate Models on Test Data

After model training was completed, we then evaluated both Decision Tree and AdaBoost models using the 10 buggy cases not used for training. For a fixed specificity of 0.99, monitor sensitivity was evaluated experimentally for each case, with each time step providing one data point used to compute the TP and FP totals. Overhead was assessed using Table 1, and alert-time was set to 0.01 s for all cases since snapshot processing considers only a single sample (with a 100 Hz sample rate in this case). Criteria were compared to the design problem described by (2), and the variable sets were subsequently compared on the basis of sensitivity and overhead.

Altogether, the results comprise a $9 \times 4 \times 2$ test matrix, with 9 bugs, 4 different variables sets (as described in Table 1), and 2 ML models: Decision Tree and AdaBoost.

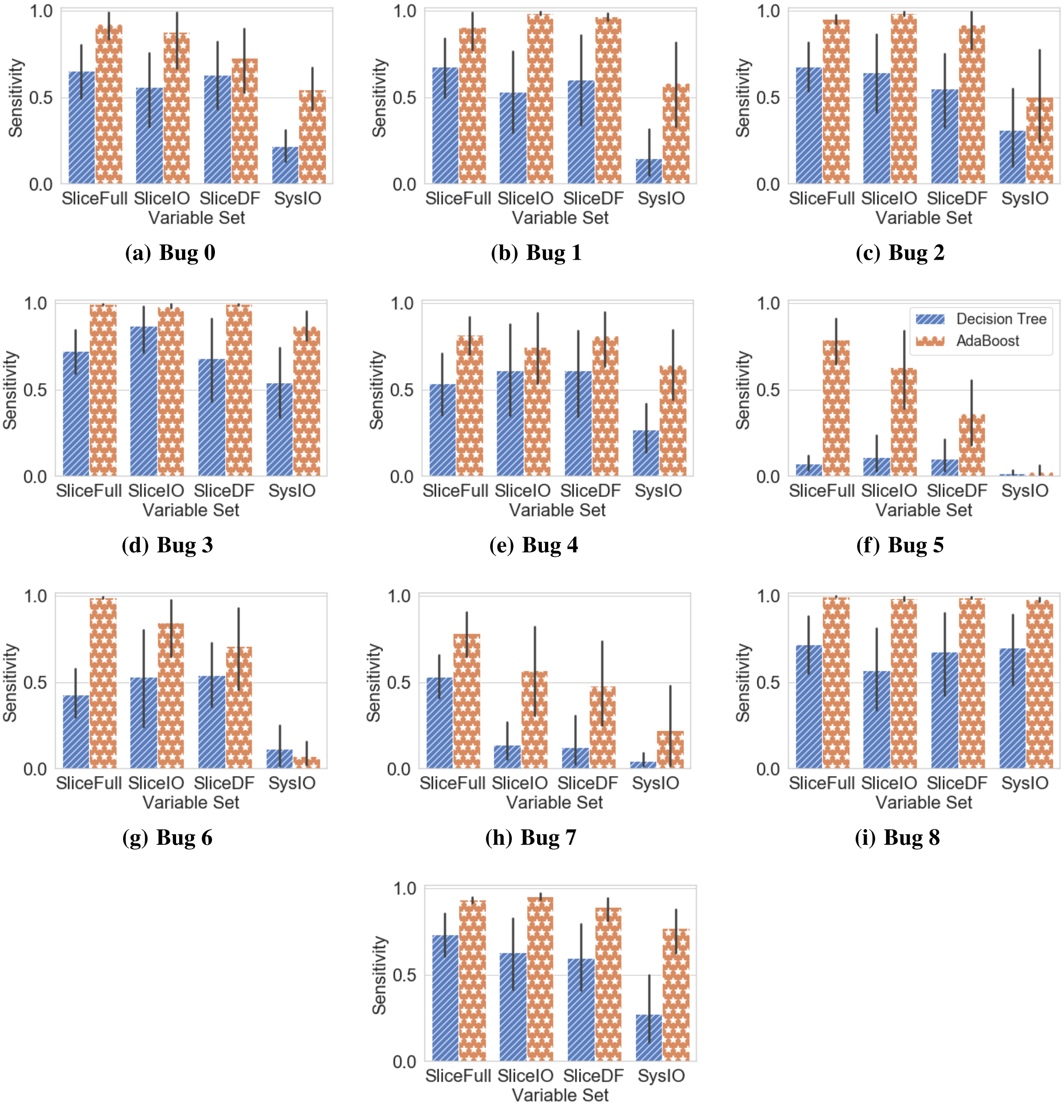
VII. Results

Sensitivity results for all nine injected bugs and the real bug are summarized in Fig. 5. Each bar plot depicts the model sensitivity of the four different variable sets for a given bug. The bars in each plot are grouped according to their variable sets in the following order: SliceFull, SliceIO, SliceDF and SysIO. Within each set of variables are the results from each of the two models: Decision Tree (blue solid bars) and AdaBoost (red polka-dot bars). The black lines at the top of each bar denote the 95% confidence interval around the mean derived from the data. Each subplot displays the average sensitivity at a specificity of 0.99 for analysis of a different bug. Another way to express specificity is to use False Positive Rate (FPR), which is defined as 1 - *specificity*. As a point of reference, if the classifier was randomly guessing, a FPR of 0.01 corresponds to an expected sensitivity of 0.01.

While model sensitivity tends to increase when training on variables from the program slice, the gains fluctuate across different bugs. A more detailed breakdown of the gains and losses in sensitivity is given in Table 2. Table 2 makes three comparisons across all bugs for the two ML models which we have chosen. The first comparison is shown in the first two columns, which records the difference in sensitivities between SliceFull versus SysIO. The other two comparisons are shown in the rest of the columns, between SliceFull verses SliceIO and SliceFull versus SliceDF.

Both SliceIO and SliceDF variable sets produced models which had worse average sensitivity (negative differences in Table 2) across all bugs in comparison to SliceFull, with SliceIO producing slightly better results. SliceIO produced models with an average loss in sensitivity of -0.049 and -0.061 for Decision Tree and AdaBoost respectively, which were higher than the loss in sensitivity from SliceDF, which were -0.054 and -0.132 respectively.

Zooming in on the performance of SliceIO and SliceDF on the injected defects, both variable sets resulted in better average sensitivity for only a small portion of the defects. For the Decision Tree and AdaBoost, learning on SliceIO produced a gain in sensitivity in only 4/9 defects (bugID: 3,4,5,6) and 2/9 defects (bugID: 1,2) respectively. While learning on SliceDF, the models produced a gain in sensitivity in only 3/9 defects (bugID: 4,5,6) for Decision Tree and only 1/9 defects (bugID: 1) for AdaBoost. AdaBoost had the largest gains in sensitivity for bug 6 at 0.92. While the



(a) **Bug 0** (b) **Bug 1** (c) **Bug 2**

(d) **Bug 3** (e) **Bug 4** (f) **Bug 5**

(g) **Bug 6** (h) **Bug 7** (i) **Bug 8**

(j) **Real Bug**

Fig. 5 Model sensitivities for two models, four variable selection methods across 9 injected bugs 5(a)-5(i) and the real bug 5(j).

Table 2 For each bug, we list gains in model sensitivities (larger numbers are better, negative numbers representing losses) for three groups and two ML models

Bug ID	SysIO vs SliceFull		SliceFull vs SliceIO		SliceFull vs SliceDF	
	DT	ADA	DT	ADA	DT	ADA
0	0.437	0.382	-0.094	-0.045	-0.020	-0.197
1	0.532	0.323	-0.145	0.081	-0.075	0.062
2	0.369	0.452	-0.032	0.029	-0.130	-0.032
3	0.183	0.126	0.145	-0.012	-0.041	-0.001
4	0.268	0.173	0.076	-0.069	0.075	-0.007
5	0.054	0.762	0.039	-0.159	0.030	-0.427
6	0.314	0.920	0.107	-0.146	0.117	-0.282
7	0.485	0.561	-0.392	-0.215	-0.406	-0.302
8	0.015	0.016	-0.146	-0.011	-0.041	-0.006
Mean	**0.295**	**0.413**	**-0.049**	**-0.061**	**-0.054**	**-0.132**
Median	**0.314**	**0.382**	**-0.032**	**-0.045**	**-0.041**	**-0.032**

majority of bugs saw an increase in model sensitivity, bug 8 had negligible gains for both models at 0.015 and 0.016. However, this negligible gain can be attributed to the high baseline sensitivity from SysIO, especially for AdaBoost with a sensitivity of 0.982.

VIII. Discussion

A. Bug Detector Performance

The goal of the design problem, as quantified in (2), is to demonstrate feasibility by delivering a snapshot bug detector that achieves a specificity of 0.99 and a sensitivity of 0.7, reducing overhead if possible. Of the monitors tested, the only implementation that achieves the required sensitivity and specificity for all bugs tested was the case using the SliceFull probe set and an AdaBoost model. (As shown in Fig. 5, no other bug detector achieved the required sensitivity for Bugs 5 and 7.) It should be noted that for most bugs the SliceDF and SliceIO probe sets performed nearly as well as SliceFull. In concept, further tuning (e.g. selecting more powerful ML models) could improve the performance of these variable sets, in which case the preferred variable set would be SliceDF, as its overhead (56 variables) is significantly less than that for SliceFull (283 variables) with minimal loss in performance (60% sensitivity or higher for all bugs). The SysIO probe set provides the lowest overhead (28 variables), but the performance of SysIO monitors is sufficiently poor that this variable set is not likely to provide successful monitoring for the bugs considered.

The results align with our assertion that variables local to a bug are more sensitive than variables extracted from elsewhere in the program. Evidence is provided from the comparison of the sensitivity of the global variable set (SysIO) to any of the local variable sets obtained from slicing. For example, if we focus on injected bugs 5 to 7, which represent three of the four bugs exhibiting waypoint overshoot, the model sensitivities shown in Fig. 5 for SysIO are very low, implying that the models are not effective at detecting these three bugs. However, when trained using the variables within the program slice, both models are markedly improved, especially for AdaBoost. In the most extreme case of an AdaBoost model trained on the SliceFull variables, sensitivity is always significantly higher than 0.7 (our target sensitivity) whereas the SysIO models exhibit sensitivity far less than 0.7, as shown in Fig. 5.

B. System Safety Implications

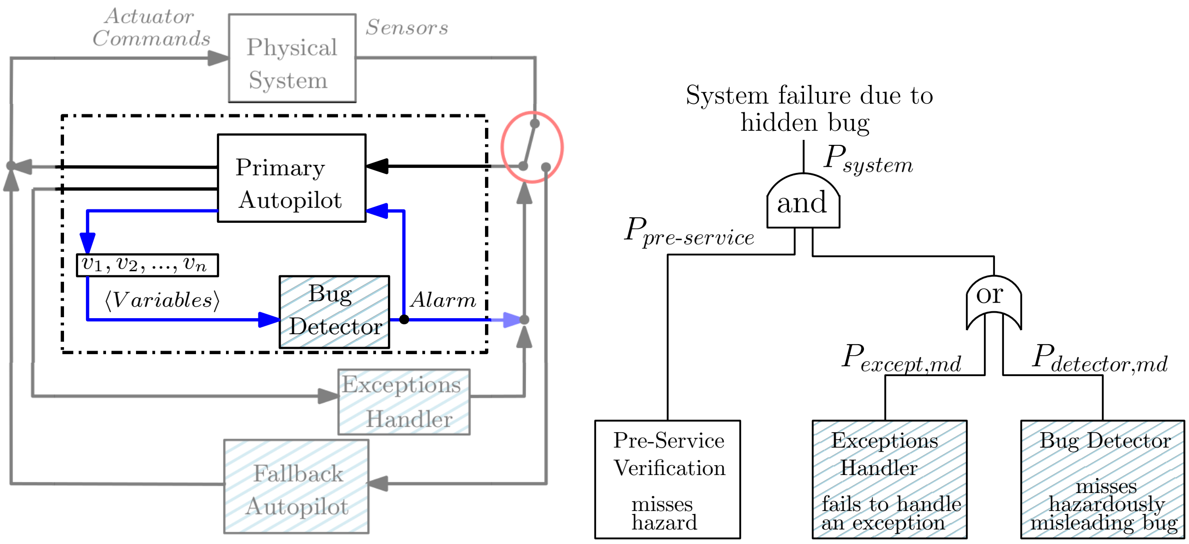
Current practice minimizes bugs in aviation software using best-practice coding and exhaustive verification (i.e. using the process specified in DO-178C [14]) prior to the software entering service. Our proposed bug detector provides an additional layer of risk mitigation, which could potentially catch hidden bugs, or might even allow a relaxation of pre-service verification without sacrificing system safety. This section puts our proposed bug detector into context, providing an overview of how our bug detector might be integrated into system-safety analysis for an unmanned aircraft.

Central to our vision is the idea that the bug detector can provide risk mitigation for a complex primary autopilot by triggering a transition to a failsafe autopilot in the event a bug is detected. Whereas the primary autopilot is assumed to be a complex, lengthy code with many advanced automation features, the failsafe autopilot is assumed to be a relatively simple, compact code with the minimal complexity needed to land the unmanned aircraft safely. Because of its small code size and reduced complexity, we believe that the failsafe autopilot can more easily be verified, preferably by formal methods but alternatively by exhaustive testing. However, we believe the complexity of the primary, full-feature autopilot will render exhaustive testing and formal methods very challenging if not impossible to apply, especially as automation software continues to grow ever more complex.

Recall that our proposed bug detector is aimed at detecting semantic bugs, when code execution diverges from the intent of the programmer. An additional monitor called an *exceptions handler* is needed to monitor for other types of bugs, like memory leaks or a complete code failure (a software crash). These types of failures are severe but obvious to detect, and they would necessarily trigger a transition from the primary to the fallback autopilot.

The relationship of the key components of the system is shown in the block diagram of Fig. 6(a). Because the bug detector and the primary autopilot are the main focus of the paper, they are shown in bold. The exceptions handler, the fallback autopilot, and the physical system (e.g. the unmanned aircraft) are shown in gray. The primary autopilot generates actuator commands that steer the physical system; the physical system produces sensor signals, which are the inputs to the primary autopilot. Variables in the autopilot (including but not limited to sensor readings and actuator commands) are inputs to the bug detector, which scans for anomalies and triggers an alarm if an anomaly is detected.

A preliminary safety analysis for the proposed system was conducted in [47] and is summarized here in brief. The basis for the safety analysis is the fault tree shown in Fig. 6(b). In analyzing the fault tree, we can model the total failure probability of the system as a collection of failures at various points of the entire safety architecture. The following



(a) Signal Flow Architecture **(b) Fault Tree**

Fig. 6 System description including 6(a) component block diagram and 6(b) fault tree.

equation expresses the total system failure-probability as an algebraic function of the component failure probabilities:

$$P_{system} = P_{pre\text{-}service} \times (P_{except,md} + P_{detector,md}) \tag{3}$$

According to Eq. (3), we can reduce fault probability of pre-service testing $P_{pre\text{-}service}$ if the fault probabilities of the bug detector $P_{detector,md}$ and exceptions handler $P_{except,md}$ are both much less than one.

The major benefit of our proposed architecture is an increase in flexibility during the design process. Currently, once exhaustive verification has been completed, there is a strong disincentive to making even minor, single-line changes to the code, as massive additional verification costs will be incurred. Consequently, the high cost of software testing has led to software being "locked" in-place once testing is completed. Any further changes are discouraged due to the requirement to re-certify, making incremental improvements difficult. We believe that introducing online monitoring can solve this problem by allowing more flexible software design, due to reduced pre-service verification requirements, with no reduction in safety.

IX. Conclusion

In this paper, we have constructed a bug monitor to detect semantic bugs. We posited that our bug monitor can detect semantic bugs by using the values of a set of variables as the data source and leveraging ML models as the method to interpret the data. We quantify the design problem in the format of an optimization problem with the goal of minimizing computational overhead while satisfying performance constraints (specificity, sensitivity, coverage, and alert-time). Due to the computational complexity of the problem, we did not pursue an optimal solution but instead used heuristics to obtain a design solution and enhance its capabilities.

We evaluated our bug detection concept using Ardupilot and have shown experimentally that program slicing enables a principled approach to identify relevant variables that enhance bug-detector sensitivity. Our results show that variables identified in the program slice can enable our ML models to perform significantly better compared to the system input and output variables. Additionally, we showed that we can largely maintain sensitivity while reducing overhead by using subsets of variables drawn from the slice.

Our analysis leads us to believe that online bug detection of autopilot software is possible. Moreover, we believe that such software can provide benefits in reducing the burdens of pre-service verification if deployed in a safety architecture as illustrated in Fig. 6(a).

Acknowledgements

This work was generously supported by the National Science Foundation through grants CNS-1329341 and CNS-1836942.

References

[1] Hawkins, A., "Boeing built a giant drone that can carry 500 pounds of cargo," *The Verge*, 2018. URL `https://www.theverge.com/2018/1/10/16875382/boeing-drone-evtol-cav-500-pounds`, [Online; accessed 2-April-2019].

[2] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C., "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, Vol. 1, No. 1, 2004, pp. 11–33. doi:\newline10.1109/TDSC.2004.2.

[3] Jenkins, A., "DJI Has Responded to Reports of Its Drones Randomly Falling Out of the Sky," *Fortune*, 2017. URL `http://fortune.com/2017/07/25/dji-spark-drones-falling-out-of-sky/`, [Online; accessed 26-June-2018].

[4] Miletich, S., "Pilot of drone that struck woman at Pride Parade gets 30 days in jail," *The Seattle Times*, 2017. URL `https://www.seattletimes.com/seattle-news/crime/pilot-of-drone-that-struck-woman-at-pride-parade-sentenced-to-30-days-in-jail/`, [Online; accessed 27-June-2018].

[5] Gibbs, S., "US aviation authority: Boeing 787 bug could cause 'loss of control'," *The Guardian*, 2015. URL `http://www.theguardian.com/business/2015/may/01/us-aviation-authority-boeing-787-dreamliner-bug-could-cause-loss-of-control`, [Online; accessed 30-November-2015].

[6] Federal Aviation Administration., "FAASTeam Notice to Operators of Rockwell Collins Flight Management Systems," , Dec. 2017. URL `https://www.faasafety.gov/SPANS/noticeView.aspx?nid=7524`, [Online; accessed 19-June-2018].

[7] Glanz, J., Creswell, J., Kaplan, T., and Wichter, Z., "After a Lion Air 737 Max Crashed in October, Questions About the Plane Arose," *The New York Times*, 2019. URL `https://www.nytimes.com/2019/02/03/world/asia/lion-`

air-plane-crash-pilots.html, https://www.nytimes.com/2019/02/03/world/asia/lion-air-plane-crash-pilots.html [Online; accessed 2-April-2019].

[8] Pasztor, A., and Tangel, A., "Investigators Believe Boeing 737 MAX Stall-Prevention Feature Activated in Ethiopian Crash," *The Wall Street Journal*, 2019. URL https://www.wsj.com/articles/investigators-believe-737-max-stall-prevention-feature-activated-in-ethiopian-crash-11553836204, [Online; accessed 2-April-2019].

[9] Dormehl, L., and Chang, L., "6 self-driving car crashes that tapped the brakes on the autonomous revolution," *Digital Trends*, 2018. URL https://www.digitaltrends.com/cool-tech/most-significant-self-driving-car-crashes/, [Online; accessed 22-June-2018].

[10] Edelstein, S., "NHTSA investigates Tesla after fatal crash while vehicle was in autonomous mode," *Digital Trends*, 2016. URL https://www.digitaltrends.com/cars/tesal-model-s-crash-nhtsa-investigation-fatal-crash/, [Online; accessed 19-June-2018].

[11] Hull, D., and Smith, T., "Tesla Driver Died Using Autopilot, With Hands Off Steering Wheel," *Bloomberg*, 2018. URL https://www.bloomberg.com/news/articles/2018-03-31/tesla-says-driver-s-hands-weren-t-on-wheel-at-time-of-accident, [Online; accessed 19-June-2018].

[12] Staff, M. I. S. R. A., *MISRA C:2012: Guidelines for the Use of the C Language in Critical Systems*, Motor Industry Research Association, 2013.

[13] Seacord, R., *The CERT C Secure Coding Standard*, SEI Series in Software Engineering, Pearson Education, 2008.

[14] Inc., R., "RTCA/DO-178C Software Considerations in Airborne Systems and Equipment Certification," Tech. rep., RTCA Inc., Washington, DC, USA, Dec. 2011.

[15] Kim, M., Viswanathan, M., Ben-Abdallah, H., Kannan, S., Lee, I., and Sokolsky, O., "Formally specified monitoring of temporal properties," *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, IEEE Comput. Soc, 1999, pp. 114–122. doi:10.1109/EMRTS.1999.777457.

[16] Miller, S., Anderson, E., Wagner, L., Whalen, M., and Heimdahl, M., "Formal verification of flight critical software," *AIAA Guidance, Navigation, and Control Conference and Exhibit*, 2005, p. 6431. doi:10.2514/6.2005-6431.

[17] Cofer, D., Gacek, A., Backes, J., Whalen, M. W., Pike, L., Foltzer, A., Podhradsky, M., Klein, G., Kuz, I., Andronick, J., Heiser, G., and Stuart, D., "A Formal Approach to Constructing Secure Air Vehicle Software," *Computer*, Vol. 51, No. 11, 2018, pp. 14–23. doi:10.1109/MC.2018.2876051.

[18] Brun, Y., and Ernst, M. D., "Finding Latent Code Errors via Machine Learning over Program Executions," *Proceedings of the 26th International Conference on Software Engineering*, IEEE Computer Society, Washington, DC, USA, 2004, pp. 480–490. doi:10.1109/ICSE.2004.1317470.

[19] Fei, L., Lee, K., Li, F., and Midkiff, S. P., "Argus: Online Statistical Bug Detection," *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering*, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 308–323. doi:10.1007/11693017_23.

[20] Baah, G. K., Gray, A., and Harrold, M. J., "On-line Anomaly Detection of Deployed Software: A Statistical Machine Learning Approach," *Proceedings of the 3rd International Workshop on Software Quality Assurance*, ACM, New York, NY, USA, 2006, pp. 70–77. doi:10.1145/1188895.1188911.

[21] Hangal, S., and Lam, M. S., "Tracking Down Software Bugs Using Automatic Anomaly Detection," *Proceedings of the 24th International Conference on Software Engineering*, ACM, New York, NY, USA, 2002, pp. 291–301. doi:10.1145/581339.581377.

[22] Chen, Y., Ying, M., Liu, D., Alim, A., Chen, F., and Chen, M.-H., "Effective Online Software Anomaly Detection," *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ACM, New York, NY, USA, 2017, pp. 136–146. doi:10.1145/3092703.3092730.

[23] Khalastchi, E., Kaminka, G. A., Kalech, M., and Lin, R., "Online Anomaly Detection in Unmanned Vehicles," *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2011, pp. 115–122. URL `http://dl.acm.org/citation.cfm?id=2030470.2030487`.

[24] Khalastchi, E., Kalech, M., and Rokach, L., "A Hybrid Approach for Fault Detection in Autonomous Physical Agents," *Proceedings of the 2014 International Conference on Autonomous Agents and Multi-agent Systems*, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2014, pp. 941–948. URL `http://dl.acm.org/citation.cfm?id=2615731.2617396`.

[25] Stracquodaine, C., Dolgikh, A., Davis, M., and Skormin, V., "Unmanned Aerial System security using real-time autopilot software analysis," *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2016, pp. 830–839. doi:10.1109/ICUAS.2016.7502633.

[26] Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y., and Zhai, C., "Bug Characteristics in Open Source Software," *Empirical Softw. Engg.*, Vol. 19, No. 6, 2014, pp. 1665–1705. doi:10.1007/s10664-013-9258-8.

[27] Sullivan, M., and Chillarege, R., "Software defects and their impact on system availability-a study of field failures in operating systems," *Fault-Tolerant Computing: Twenty-First International Symposium*, Los Alamitos, CA, USA, 1991, pp. 2 – 9. doi:10.1109/FTCS.1991.146625.

[28] Marzat, J., Piet-Lahanier, H., Damongeot, F., and Walter, E., "Model-based fault diagnosis for aerospace systems: a survey," *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, Vol. 226, No. 10, 2012, pp. 1329–1360. doi:10.1177/0954410011421717.

[29] ArduPilot Open Source Autopilot, 2017. URL `http://ardupilot.org/`, [Online; accessed 3-October-2017].

[30] Nott, G., "The Aussie open source effort that keeps a million drones in the air," *Computerworld*, 2018. URL https://www.computerworld.com.au/article/643450/aussie-open-source-effort-keeps-million-drones-air/, [Online; accessed 13-July-2018].

[31] LibrePilot, 2017. URL https://www.librepilot.org, [Online; accessed 3-October-2017].

[32] Paparazzi UAV, 2017. URL http://wiki.paparazziuav.org/wiki/Main_Page, [Online; accessed 3-October-2017].

[33] BetaFlight, 2019. URL https://github.com/betaflight/betaflight/wiki, [Online; accessed 6-May-2019].

[34] Patterson, S. M., "Microsoft's self-soaring sailplane improves IoT, digital assistants," *Network World*, 2017. URL https://www.networkworld.com/article/3225304/internet-of-things/how-microsofts-self-soaring-sailplane-improves-iot-digital-assistants.html, [Online; accessed 13-July-2018].

[35] Boeing Establishes New Autonomous Systems Program in Australia, 2018. URL http://boeing.mediaroom.com/2018-03-01-Boeing-Establishes-New-Autonomous-Systems-Program-in-Australia, [Online; accessed 13-July-2018].

[36] Applied Aeronautics., "The Applied Aeronautics Albatross Showcased at Boeing and Queensland Government Summit," , 2018. URL https://www.appliedaeronautics.com/boeing-appliedaeronautics/, [Online; accessed 13-July-2018].

[37] Lattner, C., and Adve, V., "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, 2004, pp. 75–86. doi:10.1109/CGO.2004.1281665.

[38] Chilimbi, T. M., and Ganapathy, V., "Heapmd: Identifying heap-based bugs using anomaly detection," *ACM SIGARCH Computer Architecture News*, Vol. 34, ACM, 2006, pp. 219–228. doi:10.1145/1168857.1168885.

[39] Zheng, A. X., Jordan, M. I., Liblit, B., Naik, M., and Aiken, A., "Statistical Debugging: Simultaneous Identification of Multiple Bugs," *Proceedings of the 23rd International Conference on Machine Learning*, ACM, New York, NY, USA, 2006, pp. 1105–1112. doi:10.1145/1143844.1143983.

[40] Liu, C., Yan, X., Fei, L., Han, J., and Midkiff, S. P., "SOBER: Statistical Model-based Bug Localization," *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, New York, NY, USA, 2005, pp. 286–295. doi:10.1145/1081706.1081753.

[41] Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I., "Scalable Statistical Bug Isolation," *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, New York, NY, USA, 2005, pp. 15–26. doi:10.1145/1065010.1065014.

[42] Dickinson, W., Leon, D., and Podgurski, A., "Finding Failures by Cluster Analysis of Execution Profiles," *Proceedings of the 23rd International Conference on Software Engineering*, IEEE Computer Society, Washington, DC, USA, 2001, pp. 339–348. doi:10.1109/ICSE.2001.919107.

[43] Huang, H., “Detecting Semantic Bugs in Autopilot Software by Classifying Anomalous Variables,” Ph.D. thesis, Tufts University, Medford, MA, USA, 2019.

[44] Hutchins, M., Foster, H., Goradia, T., and Ostrand, T., “Experiments of the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria,” *Proceedings of the 16th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1994, pp. 191–200. doi:10.1109/ICSE.1994.296778.

[45] Christmansson, J., and Chillarege, R., “Generation of an error set that emulates software faults based on field data,” *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on*, 1996, pp. 304–313. doi:10.1109/FTCS.1996.534615.

[46] Duraes, J., and Madeira, H., “Emulation of Software Faults: A Field Data Study and a Practical Approach,” *Software Engineering, IEEE Transactions on*, Vol. 32, No. 11, 2006, pp. 849–867. doi:10.1109/TSE.2006.113.

[47] Rife, J. H., Huang, H., and Guyer, S. Z., “Applying Sensor Integrity Concepts to Detect Intermittent Bugs in Aviation Software,” *Navigation*, forthcoming.

[48] Weiser, M., “Program Slicing,” *Proceedings of the 5th International Conference on Software Engineering*, IEEE Press, Piscataway, NJ, USA, 1981, pp. 439–449. URL `http://dl.acm.org/citation.cfm?id=800078.802557`.

[49] Weiser, M., “Program Slicing,” *IEEE Transactions on Software Engineering*, Vol. 4, No. SE-10, 1984, pp. 352–357. doi:10.1109/TSE.1984.5010248.

[50] Tip, F., “A Survey of Program Slicing Techniques.” Tech. rep., CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands, 1994.

[51] Cooper, K. D., Harvey, T. J., and Kennedy, K., “A simple, fast dominance algorithm,” *Software Practice and Experience*, 2001.

[52] Berndt, J., “JSBSim: An Open Source Flight Dynamics Model in C++,” *AIAA Modeling and Simulation Technologies Conference and Exhibit*, American Institute of Aeronautics and Astronautics, 2004. doi:10.2514/6.2004-4923.

Supplemental Material - Detecting Semantic Bugs in Autopilot Software by Classifying Anomalous Variables

The following supplemental material is intended to enhance the paper by providing additional details to support future attempts to recreate our experimental results or to run similar experiments. The supplement is organized into three sections as follows. Section SM.I gives an overview of our bug detection system designs including our instrumentation application program interface (API) and compiliation framework. Section SM.II describes an analysis of the bugs in Ardupilot that motivates our evaluation with injected bugs. Section SM.III gives further background on Bug 7062, an actual bug from Ardupilot. All relevant files for Ardupilot, our instrumentation framework, and related configuration files can be found at our `github` repository [54].

SM.I. Bug Detection System Designs

A. Oscilloscope Instrumentation API

To collect the necessary data from selected probes within Ardupilot, we have created an instrumentation API, which we named Oscilloscope (OScope). OScope collects values of variables into a uniquely-named log file during each execution of Ardupilot. OScope is implemented independent of Ardupilot in C++ and is combined into Ardupilot after lowering it first into LLVM bitcode. Using the same compilation methods, this will allow us to combine OScope into other autopilots in the future.

1. Interface

OScope contains the following functions at its external interface:

- `oscope_init` - Creates a log file with a unique name and maps it into memory using `mmap`. We chose to use a memory-mapped file instead of writing directly to disk to reduce logging latency so as to meet timing constraints.
- `oscope_start` - Starts writing to the memory-mapped file. We chose to include a specific start function because Ardupilot has its own initialization phase, which we want to bypass to reduce variability in the data being captured. We want to target the main feedback loop for Ardupilot where the software spends most of its time.
- `oscope_cleanup` - Shuts off instrumentation, unmaps the log file from memory and truncates it to the appropriate size.
- `oscope_record` - Writes the name of a variable and its value into the memory-mapped file. Due to the different data types in the code, we create a separate record function for each data type: `float`, `double`, `int8`, `int16`, `int32` and `int64`. We will use the appropriate record function according to the type of the variable being instrumented.

OScope does not modify the Ardupilot source code directly. Instead OScope functions are inserted indirectly in different locations in Ardupilot (in its LLVM IR bitcode form) by our `instrumenter` (as discussed in Section SM.I.B, below).

`oscope_init()` is inserted as the first instruction of the first basic block of `main()` and `oscope_cleanup()` is inserted as the last instruction before the program returns from `main()`. We chose this approach because we want to avoid variations from initialization code such as object constructors that may execute before `main()`. `oscope_start()` is inserted into a function called `loop()`. After the set of relevant variables are located, a call to the appropriate `oscope_record` function is inserted immediately after these variables.

2. Snapshots

A snapshot is a list of key-value pairs which maps a variable to its value at a specific time. Because processing is conducted sequentially on the same processor, no two variables are updated at *exactly* the same instant in time, even though control engineers often model computers as updating variables at precisely the same time. By identifying snapshots, we decompose time into blocks with a relevant granularity for the autopilot software. All variables written at approximately the same time (within an 0.01 s period in our case) are grouped together into the same snapshot. Formally, let S_t represent the snapshot at time t and let P be the set of variables that have been selected for monitoring. We generate the snapshot S_t by constructing a list of key-value pairs (p_i, v_i), where $p_i \in P$, v_i is the value of variable p_i and $1 \leq i \leq |P|$. As the autopilot executes, the bug detector will continuously record data from each variable p_i and append the new data to the log file, thereby creating one snapshot after another contiguously.

One of our design goals for the log file is to capture the state of all the monitored variables into a snapshot. A naive implementation will always keep an entire snapshot in memory and whenever any variable updates its value, the bug detector can write the entire snapshot to the log file, thus appending the log file at roughly 100 Hz. This has the benefit of easing the burden of data processing, but has the disadvantage that this type of logging generates a log file of several hundred gigabytes for a 10-minute real-time experiment. The size of just one log file presents a large barrier to downstream data analysis since we must process many log files. For a given bug, we typically ran 30 trials, which would have resulted in tens of terabytes of storage required for analyzing a single bug. This amount of data quickly exceeded our practical limits for data transfer and storage when considering many bugs. As a result, we sought to reduce the size of each log file.

We approach this hurdle by observing that many variables do not change as quickly as the sampling frequency of 100Hz. For instance, if a variable is a flag, it has the same value repeated through potentially many snapshots. This generates a large amount of unnecessary data if we write out snapshots containing a value for every single probe. Therefore, we logged a new entry for a variable *only* when its value changes. Currently the format of the log file is: *time_since_start*, *variable_name*, *variable_value*. This method has the disadvantage that it takes more work to rebuild the snapshot, albeit off-line since the log file is now a linear trace instead of a snapshot. However, doing more work off-line is an acceptable trade-off. This results in log files that are smaller by an order of magnitude.

B. Compilation Pipeline

The `slicer` and the `instrumenter` work in concert to modify an incoming Ardupilot bitcode file before compilation to binary, as illustrated in Fig. 7. The purpose of the `slicer` is to extract a set of variables to instrument from the autopilot according to a specific policy. Examples of such policies include: all floating point variables, all variables located in the predicate of `if` statements and all variables located in the program slice. Each policy describes a different algorithm that extracts the pertinent variables. In comparison, the `instrumenter` only has one purpose, which is to add calls to the OScope API at a set of variables identified for instrumentation. Due to this difference, we have intentionally divided the two tools as separate and stand-alone components in the compilation pipeline in order to make the overall software architecture easier to develop and debug.

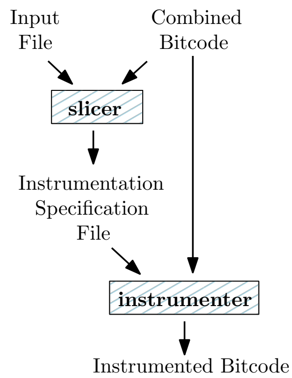


Fig. 7 Details of compilation pipeline.

As shown in Fig. 7, the `slicer` and the `instrumenter` are connected through the input and output files that they generate. The `slicer` takes two inputs, an LLVM bitcode file and an input specifications file, and produces an output specification file containing the list of sites for instrumentation. Each line of this output specification contains enough information for the `instrumenter` to find the appropriate instruction, such as the function name, the relative offset of the bitcode, etc. The output specification file serves as the input for the `instrumenter`, which uses these specifications to instrument the bitcode.

```
ap_plane_bug7062/libraries/AP_Baro/AP_Baro.cpp _ZN7AP_Baro6updateEv 28 516
ap_plane_bug7062/libraries/AP_Baro/AP_Baro.cpp _ZN7AP_Baro6updateEv 34 518
ap_plane_bug7062/libraries/AP_Baro/AP_Baro.cpp _ZN7AP_Baro6updateEv 120 534
ap_plane_bug7062/libraries/AP_Baro/AP_Baro.cpp _ZN7AP_Baro6updateEv 143 536
ap_plane_bug7062/libraries/AP_Baro/AP_Baro.cpp _ZN7AP_Baro6updateEv 153 537
ap_plane_bug7062/libraries/AP_Baro/AP_Baro.cpp _ZN7AP_Baro6updateEv 182 541
ap_plane_bug7062/libraries/AP_Baro/AP_Baro.cpp _ZN7AP_Baro6updateEv 202 544
ap_plane_bug7062/libraries/AP_Baro/AP_Baro.cpp _ZN7AP_Baro6updateEv 223 546
ap_plane_bug7062/libraries/AP_Baro/AP_Baro.cpp _ZN7AP_Baro6updateEv 292 564
ap_plane_bug7062/libraries/AP_Baro/AP_Baro.cpp _ZN7AP_Baro6updateEv 295 566
ap_plane_bug7062/libraries/AP_Baro/AP_Baro.cpp _ZN7AP_Baro6updateEv 297 567
ap_plane_bug7062/libraries/AP_Baro/AP_Baro.cpp _ZN7AP_Baro6updateEv 308 567
ap_plane_bug7062/libraries/AP_Baro/AP_Baro.cpp _ZN7AP_Baro6updateEv 311 569
```

Fig. 8 An example of the output specification that the `slicer` generates from the LLVM bitcode.

Fig. 8 illustrates an example of the output specification from the `slicer`. Each line in the specification contains four columns to locate an LLVM Intermediate Representation (IR) instruction in the bitcode: a file name, a function name, the offset of the LLVM IR instruction relative to the start of the body of the function, and the original line number in the source code. There are two design goals to the output specification file. The primary design goal is to enable the `instrumenter` to find the instrumentation site without any ambiguities. The secondary design goal is to use the output specification as another check-point to verify the correctness of the `slicer`. For a particular policy, if the `slicer` has identified *all* of the relevant variables then it is deemed to be correct.

An important implementation detail is that the name of a function refers to a name that has been garbled by the compiler and not to the original function name in the Ardupilot source. This process of garbling is called "mangling", which is a necessary step in the compilation process to turn function names into unique identifiers. While it would have been more user-friendly to have a human-readable function name in the output specification, unfortunately this information is lost when the C++ preprocessor mangles all function names. Therefore, the mangled function names are the only references we can use as a key to refer to a function. It is also worth noting that the code references are numerical rather than text based. Using a numerical offset of the LLVM instruction instead of the actual text representation of the instruction precludes the `instrumenter` from having to do text matches and allows it to find each instruction through a linear scan.

C. Data Logging

The `logprocessor` was created as a tool to recreate the snapshots from the data file which OScope generated during Ardupilot flight simulation. Due to both the large number of log files and the large size of each log file, each log file is compressed to save disk space. The `bzcat` utility was run on the log file to decompress the log files as a stream, and this data stream served as the input to the `logprocessor`.

The `logprocessor` reads the decompressed log file from standard input and reconstruct the snapshot in two passes. We require two passes because of the streaming nature of the input, we don't know ahead of time how many variables will be included in the log file. Therefore, on the first pass, the `logprocessor` tracks the set of unique variables present in the log file and writes the result to a file. On the second pass, the `logprocessor` uses the result from the first pass and the log file to reconstruct the snapshots.

There are two hurdles in the implementation of snapshots. The first hurdle is that as the autopilot executes and data is continuously logged to the log file, it becomes quite difficult to separate one snapshot from another. The second hurdle is that the probe updates are not uniform. Even within one snapshot, there are variables which do not update their values at all or update their values more than once. We discuss each hurdle and its solution in turn below.

To address the first hurdle of how to separate one snapshot from another in the contiguous log file, we add a variable named `loopCount` in the source code. The `loopCount` variable is placed at the top of the `loop()` function and incremented every time the function executes. The variable acts a sentinel in the log file, so that downstream analyses can partition on it, to separate one snapshot from another in the data.

To tackle the second hurdle of nonuniform variable updates, we make several assumptions. If variable p_i does not update its value v_i in snapshot S_t, then we assume that variable p_i still retains the same value as in a previous snapshot S_{t-k}, where $1 \leq k \leq t-1$. At time $t = 0$ we set all the values of all the variables to 0. To facilitate the lookup of the previous value, we always keep the value of p_i from the previous snapshot. For variables which update multiple times within one snapshot, we only take into account the last update and ignore the rest.

SM.II. Ardupilot Bugs Analysis

In order to design injected bugs that are representative of real bugs, we need an understanding of the types of bugs which have been reported both in the wider literature and in Ardupilot. This Section provides supporting rationale to explain why the form of the injected bugs was chosen and why we believe that form is representative of real bugs that have previously occurred in the development of Ardupilot.

One of the first studies of software bugs came from Sullivan and Chillarege [27, 55], where they created the Orthogonal Defect Classification (ODC) system to classify defects found in an IBM database and operating system. Subsequent work by Christmansson and Chillarege [45] used ODC analysis to classify defects recorded in the field in order to generate a set of representative injected bugs. As software has gone through drastic changes since 1996, one may presume that these defect distributions are no longer applicable. Recent work [46] however has reapplied the same ODC analysis to desktop software and found that the distribution of defects has not changed.

Table 3 Distribution of defects in the Ardupilot Bug Database

Defect Category	# Defects
Assignment	19
Checking	21
Interface	1
Algorithm	26
Function	10
Timing/Seria.	7
Other	28
Total Defects Examined	**112**

We have applied the ODC analysis to the Ardupilot Database to check whether the defect distribution in Ardupilot resembles the defect distributions observed in prior studies. We examined 112 defect reports in the Ardupilot Bug Database spanning from April 2014 to January 2018. A detailed breakdown of all the defects is listed in Table 3. In this table we included only reports with a “Bug” label, with an effect on any aircraft platform, and with the “Closed” status. The reason that we chose only “Closed” defects is because only these defects contain enough details for an appropriate classification. Out of the 112 defects, only 84 were determined to be defects that fit the six categories we list below. The remaining 28 defects did not fit the ODC categorization scheme, so were excluded here (though they may be worth considering in more detail in the future). Examples of excluded bugs include build-system defects where the code fails to compile. The six ODC defect categories are as follows (reproduced from [46]):

1) **Assignment** - Values assigned incorrectly or not at all.
2) **Checking** - Missing or incorrect validation of data or incorrect loop or conditional statements.
3) **Algorithms** - Incorrect or missing implementation that can be fixed by (re)implementing an algorithm or data structure without the need of a design change.
4) **Interface** - Errors in the interaction among components, modules, device drivers, call statements, or parameter lists.
5) **Function** - Affects a sizable amount of code and refers to capability that is either implemented incorrectly or not implemented at all.
6) **Timing/Serialization** - Missing or incorrect serialization of shared resources.

Table 4 shows the results of our defect analysis on the Ardupilot database in comparison to the distribution of faults found in both Christmansson [45] and Duraes [46]. Each column displays the number of faults and their associated percentages. Entries with "–" indicate that there was no data.

Table 4 The defect distribution of Ardupilot in comparison to the distributions found in other defect classification studies

ODC Defect-Type	Christmansson [45]	Duraes [46]	Ardupilot
Assignment	78 (19.1%)	143 (21.4%)	15 (22.6%)
Checking	62 (15.2%)	167 (25%)	21 (25%)
Interface	29 (7.1%)	49 (7.3%)	1 (1.2%)
Algorithm	154 (37.8%)	268 (40.1%)	26 (31%)
Function	31 (7.6%)	41 (6.1%)	10 (11.9%)
Timing/Seria.	54 (13.2%)	–	7 (8.3%)
Total Defects	**408**	**668**	**84**

Our results indicate that for the top three categories of bugs (Algorithm, Assignment and Checking), Ardupilot has a similar distribution in comparison to the other two studies. Even though the Algorithm defect category makes up a slightly smaller fraction of bugs for Ardupilot than observed in prior studies, it is clear that Algorithm defects still dominate and represent the *largest percentage* of bugs across all three studies. Following closely behind are Checking and Assignment defects. The fraction of bugs in the Function and Interface categories are small for Ardupilot, similar to observations in prior studies. Though the precise percentages appear slightly different for Ardupilot as compared to other studies, these values should not be over-interpreted, because the range of error is similar to the differences in mean.

Timing/serialization defects were not present in Duraes's study and therefore that entry is marked "–". Even after taking into account the defect categories with no data, the distribution of bugs across all three columns in Table 4 is remarkably similar.

In his work, Duraes performed a deeper analysis into software defects, which further guided our injected bug design. He categorized all the defects according to the programming language constructs that were either missing, wrong or extraneous and cross-referenced them by their ODC defect types. Table 5 is a reproduction of Duraes' results that shows a refined categorization of bugs listed as *Checking* or *Algorithm* bugs in Table 4. Surprisingly, we find that 85.9% (or 281 out of 327 defects listed in Table 5) relate to `if` statements in some way. Moreover, most of the defect types that affect `if` statements result in code being omitted during run-time.

Table 5 The top 65% of bugs observed in Duraes' study [46] categorized by programming language constructs that were either missing, wrong or extraneous

	Fault Type	# Faults	ODC types				
			ASG	CHK	INT	ALG	FUN
Missing	`if` construct plus statements	71				✓	
	`AND` sub-expr in expression used as branch condition	47		✓			
	Function call	46				✓	
	`if` construct around statements	34		✓			
	`OR` sub-expr in expression used as branch condition	32		✓			
	Small and localized part of the algorithm	23				✓	
	Variable assignment using an expression	21	✓				
	Functionality	21					✓
	Variable assignment using a value	20	✓				
	`if` construct plus statements plus `else` before statements	18				✓	
	Variable initialization	15	✓				
Wrong	Logical expression used as branch condition	22		✓			
	Algorithm - large modifications	20					✓
	Value assigned to variable	16	✓				
	Arithmetic expression in parameter of function call	14			✓		
	Data types of conversion used	12	✓				
	Variable used in parameter of function call	11			✓		
Extra	Variable assignment using another variable	9	✓				

The design of our injected bugs began with a careful review of this defect data in previous studies. Taken together, the bug logs for Ardupilot and prior studies like Duraes' clearly suggest that conditional statements are at the root of a great many software defects, most often resulting in a block of code within an `if` statement being omitted. Using these observations, we created a number of branch-statement defects that force the program to not execute certain branches in `if` statements. These synthetic bugs are recorded on `diff` files and can be applied directly to the source code. The `diff` files are found in our `github` repository [54]. Bugs were injected one at a time, so there was never more than one real or synthetic bug active simultaneously. All of the synthetic defects introduced involved a conditional statement which was erroneously disabled, so that only one branch of that `if` statement was accessible. The prevalence of conditional logic defects in the Ardupilot database and in Duraes' work suggest that our injected bugs are representative of real semantic bugs that one might expect to encounter in development and verification of an autopilot. The nature of conditionals is that some branches are rarely active, and so defects in those branches may be difficult to detect in pre-service verification, even after an extensive testing campaign.

SM.III. Bug 7062

In related work [43], we conducted detailed studies on three bugs from the Ardupilot database: Bug 2835 (sudden impulsive change in aircraft pitch when tracking waypoints through a sharp right turn), Bug 6637 (failure to follow circular loiter trajectory), and Bug 7062 (failure to transition from vertical climb to forward flight for a vertical takeoff and landing, or VTOL, aircraft). In this paper we focus on a compact region of the Ardupilot code that includes only one of real bugs, Bug 7062. The analysis in the paper focuses on this bug (and complementary analysis of synthetic bugs) in order to characterize the performance of the proposed bug detector. For testing purposes, we reproduced Bug 7062 in simulation and have defined a flight scenario that was likely but not guaranteed to trigger the bug of interest. Below we discuss Bug 7062 in further detail.

A. Overview

A quad plane is a hybrid aircraft, a fixed-wing plane with 4 horizontal rotors attached to provide a VTOL capability. In comparison to a tilt-rotor craft like the V-22 Osprey [56], the quad plane has the advantage that it can shut off rotors instead of tilting its rotors to convert them into propulsors. However, having rotors on a fixed-wing plane introduces additional layer of complexity during takeoffs, and it is this complexity that resulted in Bug 7062. Bug 7062 manifests during the duration of VTOL takeoff and can be described as a failure to transition from rotor-based hovering to forward flight during a vertical takeoff maneuver. In the words of the developers who fixed the bug [57]:

> *If a fixed-wing loiter is initiated and transition is incomplete and the current altitude is higher than the desired altitude, the aircraft attempts to lower the forward thrust in order to descend... but if this prevents the vehicle reaching transition speed, then the VTOL motors can prevent it [the aircraft] from losing altitude. This can lead to a situation where transition speed is never reached, and altitude continues to climb.*

B. Root Cause

The quad plane takes off using its rotors. Once it reaches the target altitude, it can transition between VTOL flight and fixed-wing flight. The necessary condition for this transition to take place in Ardupilot is that the plane must achieve sufficient forward velocity. This is necessary so that the plane has enough lift when the horizontal motors are switched off. If a flight scenario contains a waypoint that is lower in altitude than the current altitude, Bug 7062 manifests. In order to get to the new lower waypoint, the autopilot needs to descend and so it decreases throttle. At the same time however, the autopilot needs to increase forward velocity in order to transition out of climb to forward flight. This tension leads the throttle to oscillate back and forth. The autopilot never obtains the necessary forward velocity and continues to climb.

We summarize what is occurring inside the code with a state transition diagram as depicted in Fig. 9. There are four pertinent states involved in the transition. Whenever a successful transition is completed, the autopilot ends in the *Done* state. When the transition begins, the autopilot starts in the *Start* state. The usual path (with the edges colored in black) that the autopilot takes is *Start* → *Wait* → *Timer* → *Done*. The code also provides a way to go from *Wait* → *Done* in certain situations, but they are uncommon. The edges colored in red are back edges in this graph and they are the problem. They allow the system to go back from *Timer* to *Wait* and the system will remain in *Wait* indefinitely.

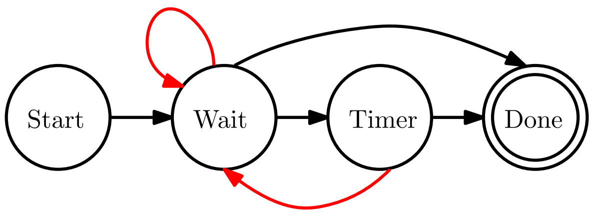


Fig. 9 State transition diagram for Bug 7062.

The root cause for the back edges is that when the system goes from *Wait* to *Timer*, the autopilot starts a timer because the developers expected the quad plane to build up enough forward velocity to finish the transition within a specific amount of time. However, when the quad plane is oscillating its throttle in trying to meet two competing demands, it exceeds the expected transition time. The code inadvertently provides for a way for the state to go back to *Wait* by default and thereby this creates an infinite loop. The end result is that the quad plane continues to climb and is stuck in takeoff mode.

Since the issue is that the autopilot is trying to meet two competing demands, the autopilot could never satisfy both. So in order to fix Bug 7062, Ardupilot developers decided to remove any attempts from the autopilot to control forward velocity during VTOL.

References

[54] JAIS Submission File Repository, 2019. URL https://github.com/RedlineResearch/HuangJAIS2019Materials, [Online; accessed 11-June-2019].

[55] Chillarege, R., Bhandari, I. S., Chaar, J. K., Halliday, M. J., Moebus, D. S., Ray, B. K., and Wong, M.-Y., "Orthogonal Defect Classification - A Concept for In-Process Measurements," *IEEE Transactions on Software Engineering*, Vol. 18, No. 11, 1992, pp. 943–956.
doi:10.1109/32.177364

[56] Mizokami, K., "V-22 Osprey | Bell Boeing V-22 Osprey History," *Popular Mechanics*, 2019. URL https://www.popularmechanics.com/military/aviation/a26242129/v-22-osprey-tiltrotor-bell-boeing/, [Online; accessed 13-April-2019].

[57] ArduPilot Github - Issue #7062, 2018. URL https://github.com/ardupilot/ardupilot/issues/7062, [Online; accessed 16-July-2018].