Applying Sensor Integrity Concepts to Detect Intermittent Bugs in Aviation Software

Jason H. Rife, Hu Huang, and Sam Z. Guyer, *Tufts University*

CORRESPONDING AUTHOR:
Jason Rife
jason.rife@tufts.edu

CITATION
Rife, Jason H., Huang, Hu, Guyer, Sam Z., "Applying sensor integrity concepts to detect intermittent bugs in aviation software," *NAVIGATION, Journal of The Institute of Navigation*, Vol. 66, No. 3, Fall 2019, pp. 603-619. https://doi.org/10.1002/navi.322

FINANCIAL SUPPORT:
National Science Foundation (grants 1329341 and 1836942).

ABSTRACT

This paper assesses potential benefits of online integrity monitoring for aviation software. Today, aviation software safety is assessed entirely via verification and validation conducted before deployment, largely through exhaustive testing. Though software anomalies (or *bugs*) do occur, system fault trees do not assign fault probabilities to software components. Moreover, online signal monitoring is not generally applied to detect bugs. As software complexity increases, and as interactions between software and hardware system components become more complex, it is prudent to consider whether exhaustive pre-service software testing is sufficient to maintain system safety, and whether safety analyses used for hardware components (e.g. fault trees and monitoring) might be applied to software components. The contribution of this paper is to propose an architecture for online bug monitoring and quantify its potential. The proposed concept has significant potential impact for low-cost, autonomous unmanned aircraft systems, where cost drivers prohibit exhaustive pre-service verification.

INTRODUCTION

In safety-of-life aviation applications, integrity monitors are critical for detecting navigation-system faults that might otherwise cause hazardously misleading information. To protect aviation applications reliant on the Global Navigation Satellite System (GNSS), for example, engineers have developed a number of GNSS integrity-monitoring systems including RAIM, GBAS, and SBAS [1]-[4]. These monitoring capabilities inherently acknowledge that anomalous events can occur, even in technically well-designed systems.

By contrast, in deploying aviation-software systems, software is assumed to be perfect once verified. This assumption implies the algorithm structure is fully specified in advance and implemented exactly. Although the notion of exhaustive software verification is reasonable for simple software programs, verification complexity becomes daunting as software complexity increases, particularly for aviation systems and other cyberphyiscal systems in which software and hardware components interact. Indicative of the skyrocketing complexity of aviation software, the number of lines in flight-control codes have increased by an order of magnitude each decade [5],[6]. Resulting complexity has substantially slowed development of innovative conventional aircraft and emerging unmanned aerial systems (UAS) [7],[8]. Serious software anomalies have also caused problems for large and complex commercial aircraft [9].

One active branch of research that seeks to address the challenges of verifying complex code is the field of *formal methods* [10]-[13]. Formal methods apply specification-based approaches to generate automated proofs of how a code will perform. Though this approach has promise, it has proved impractical to date because of rigid constraints on the software development process, including the algorithms, languages and syntax used. Moreover, writing and checking formal specifications is extremely difficult and time-consuming. Even using today's technology, a software verification specialist must be hired. Other longer term issues include the possibility of errors appearing in the specification itself and inflexibility, which makes iterative updates to the specification extremely difficult.

Given that it is difficult to deploy formal methods for large-scale aviation software (which is often written in general-purpose languages like C), we propose an alternative concept for achieving high levels of safety without overly constraining aviation software developers. Specifically, in this paper, we explore the potential utility of online bug monitoring for simplifying verification. The concept is that aviation software would consist of a large flight control code complemented by a small, failsafe kernel. The large flight control code would be verified to a reasonable degree, but only the failsafe kernel would be verified exhaustively (or perhaps designed using formal methods). While the primary flight control code would be active in nominal operations, a transition to the kernel would occur if a potentially hazardous bug were detected during operation; the failsafe kernel would have just sufficient capability to navigate the aircraft to safety in that event. Triggering the failsafe kernel necessitates an effective real-time bug-monitoring capability. As we will explore in this paper, if such a bug monitor achieves sufficient levels of performance, the overall software architecture delivers a high level of integrity, compatible with safety-of-life aviation applications. Importantly, the proposed architecture would greatly relax pre-operation verification requirements, allowing for more rapid development of aviation software and enabling the use of a wide range of programming languages,

data structures, and modern algorithms (including machine learning). The reduced requirements for pre-service verification would promote more nimble software design processes, including iterative design methods (like *agile* design [14]-[16]), which are difficult to implement given the high cost of re-verifying existing software using conventional verification standards [17],[18].

The main contribution of this paper is to propose a system architecture and corresponding safety case that exploit online bug monitoring to relax requirements for pre-service verification without sacrificing overall system safety. Though we have previously introduced concepts for bug monitors [20], the safety case for such monitors has not previously been considered.

The remainder of the paper is structured as follows. First, we develop a high-level fault tree that captures the potential contribution of a bug-monitor to overall system safety. Second, we describe a prototype bug-monitor that we have implemented using an open-source flight controller called Ardupilot. Third, we identify key performance metrics that can be used to evaluate the monitor's performance, and we develop a mathematical approach to quantify those performance metrics. Fourth, we explore the design space for the monitor using a very simple probabilistic model of monitor operation. In discussing this design space, potential design targets for a monitor are identified. A brief summary concludes the paper.

PROPOSED SYSTEM ARCHITECTURE

This section defines a system architecture and an associated fault tree, in order to analyze how a bug monitor might relax requirements for pre-service verification while preserving overall system safety. Three primary mitigations for bug-induced faults are proposed: pre-service verification, crash recovery, and bug monitoring. A fault-tree including all three mitigations is shown in Fig. 1.

As per current practice for verifying aviation software, detailed in DO-178 [17], we envision pre-service verification to be the foundation for mitigating bug-induced faults. Pre-service verification is intended to catch most bugs in the code, and in particular those bugs that occur during nominal operations. This process is assumed to consist of extensive run-time testing on compiled code, starting with testing of software interfaced with simulated hardware and concluding with validation of the software integrated with the full physical system.

Unlike conventional verification for aviation software, a finite probability of a failure due to a bug is specified on the system fault tree. An analysis of the number and type of verification tests will be needed to determine whether a pre-service verification campaign can meet the target probability that a hazardous bug remains after verification. If online bug detection is sufficiently

reliable, then it may be possible to tolerate a greater likelihood of a latent bug remaining in the code. This mitigation concept implies that rare bugs – at least those escaping initial verification efforts – are inherently somewhat random in nature. We assert that such a probabilistic model is appropriate for difficult-to-find-bugs, which appear and disappear with no discernible pattern. (Such bugs, which seemingly vanish when probed, are common enough that programmers have coined the name Heisenbugs [19], a humorous reference to the Heisenberg Uncertainty Principle.)

As an example of possible mitigation, Fig. 1 shows how a total system failure rate of 10^{-9} per arbitrary unit of time can potentially be achieved by combining a bug monitor (missed-detection probability of 10^{-5}) and a moderate level of verification (maximum allowed probability that a hazardous bug remains post-verification of 10^{-4}). For a deployed system, the failure rate would need to be specified for a given period of time (e.g., as the allowed rate of bug faults per hour); however, at this preliminary stage in development of bug-monitor concepts, it is too early to settle on a particular time unit. As such, all probabilities in this paper are written without a specified time unit.

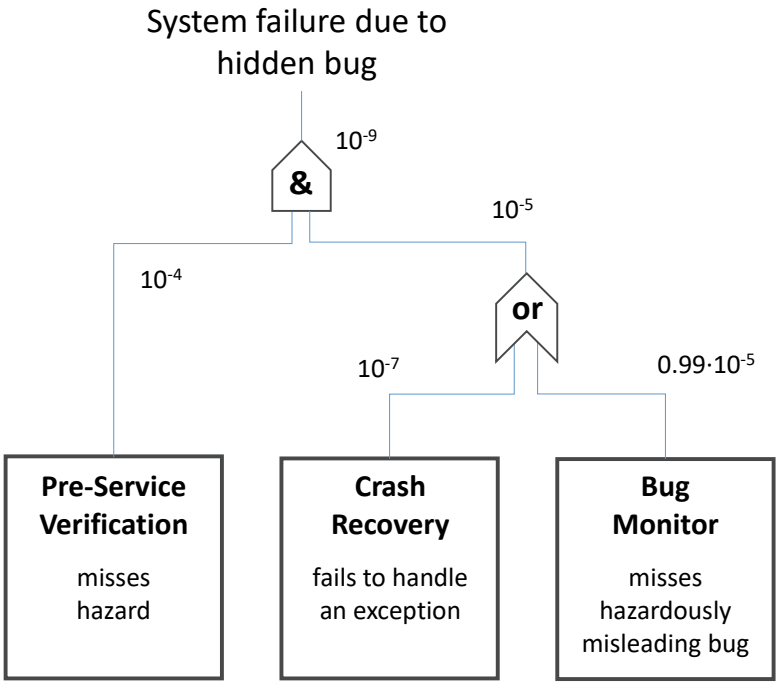

Fig. 1. Fault tree for proposed bug mitigation architecture. Example fault-probabiliy targets per unit time are shown, as one vision for how the risk of a bug causing a system failure might be limited to a maximum probability of 10-9 per unit time.

On closer examination, Fig. 1 describes a safety architecture consisting of three levels of risk mitigation for bugs, which are complemented by a pair of autopilots, as illustrated in Fig. 2. The three levels of risk mitigation include: offline verification

that occurs before the system enters service, online crash recovery, and an online bug monitoring. The distinction between crash recovery and bug monitoring acknowledges that rare, essentially random bugs manifest in one of two ways – as dramatic software crashes or as more subtle corrupted variable values. First, in the case of a software crash, such as a segmentation fault, an exception handling routine would have responsibility to switch control from the primary control code to the failsafe. Though such bug-induced faults are potentially disastrous, they have the virtue of being easy to detect and therefore mitigate. Second, in the case of a corrupted variable, a bug monitoring routine would have responsibility to switch control to the failsafe. Though a system-fault due to corrupted-variable values may take somewhat longer to develop than one due to a software crash, the corrupted-variable case is much harder to detect and mitigate. As such, the emphasis of this paper will be on analyzing the performance of the bug monitoring routine.

Returning to Fig. 1, it is clear that any bug that evades pre-service verification must be caught by either crash recovery or bug monitoring. The missed-detection probabilities are therefore combined with an OR in the figure. For a bug to harm the system, it must be missed by pre-service verification AND monitoring. The upshot is that, at least in this example, the requirements for pre-service verification are relaxed by five orders of magnitude by introducing online bug mitigation.

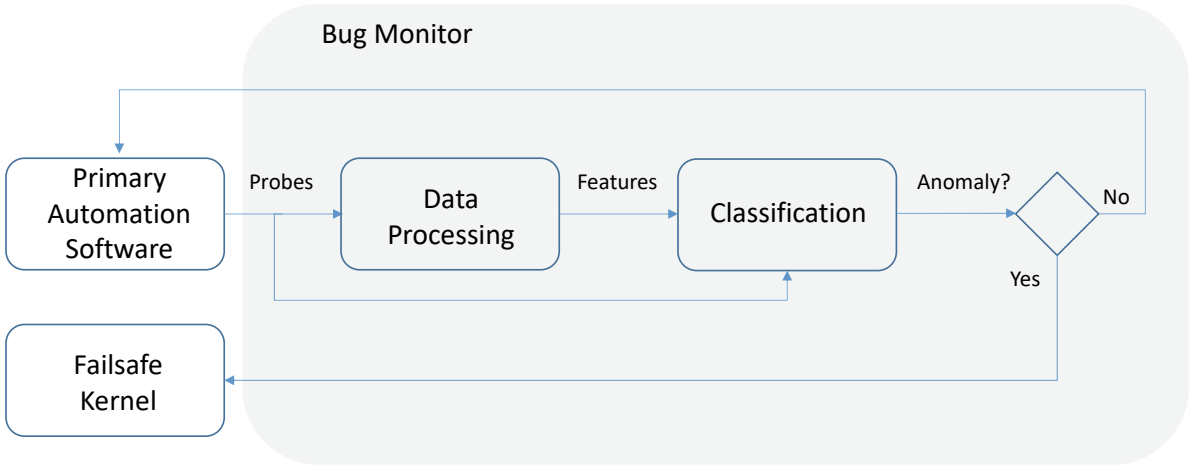

Fig. 2. Monitor block diagram

An astute reader may be concerned about additional integrity risk incurred when an alarm is issued by crash recovery or bug monitoring. Because the failsafe kernel is assumed to be verified to an exceptionally high standard, the probability of a bug occurring in the failsafe kernel is assumed to be zero, and so the fault-tree includes no additional integrity risk associated with true-positive alarms. This model is justified by the idea that the failsafe kernel is a very small code that provides just enough

functionality to safely land an aircraft and no more; because the failsafe kernel is much less complex than the primary autopilot, it is assumed that the failsafe kernel can be exhaustively verified or be constructed with formal methods such that it is guaranteed correct-by-construction. Thus, the bug monitor is only needed for the much larger, more complex primary autopilot.

Neither transitioning nor operating the failsafe induces added integrity risk; however, it must be acknowledged that transitioning to the failsafe entails interruption of service. In other words, alarms from crash recovery AND bug monitoring introduce continuity risk, as shown in Fig. 3. Continuity breaks are caused both by true and false alarms.

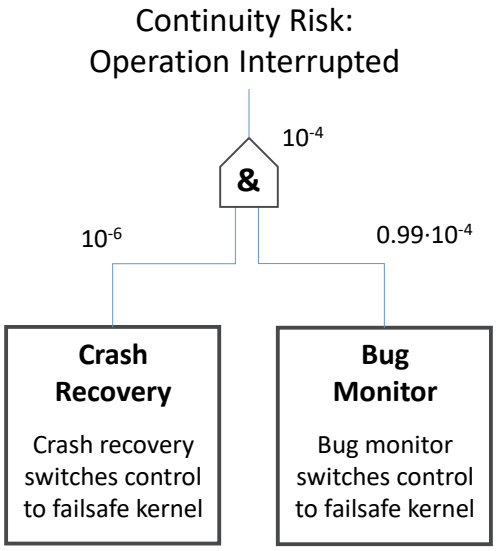

Fig. 3. Continuity risk for proposed architecture

An important question is: can acceptable levels of integrity and continuity risk be achieved by such an architecture? In order to evaluate potential performance, it is useful to convert the fault-tree in Fig. 1 into an equation, where the system-failure risk due to a software fault is described by the following series of conditional probabilities.

$$P\{\text{system failure}\} = P\{\text{bug causes failure} \mid \text{failsafe trigger fault, verification fault}\} \\ P\{\text{failsafe trigger fault} \mid \text{verification fault}\}\, P\{\text{verification fault}\} \tag{1}$$

This equation can be simplified slightly by assuming conservatively that any software bug is capable of causing a system failure, in which case the first probability above is one. Making this conservative assumption, introducing P_{VF} as a shortened notation for $P\{\text{verification fault}\}$, and introducing P_{LOI} – where LOI stands for *loss of integrity* – as a shortened notation for $P\{\text{system failure}\}$, equation (1) becomes:

$$P_{LOI} = P\{\text{failsafe trigger fault} \mid \text{verification fault}\} P_{VF} . \quad (2)$$

The probability that the failsafe does not trigger can be expanded, noting from Fig. 1 that the failsafe is triggered either by crash recovery *or* bug monitoring. This "or" condition is represented by addition of relevant probabilities:

$$P\{\text{failsafe trigger fault} \mid \text{verification fault}\} \approx P_{CR} + P_{FN} . \quad (3)$$

Here P_{CR} represents the probability that crash recovery fails and P_{FN}, that the bug monitor fails. The abbreviation *FN* stands for false-negative. Further simplification is possible if we assume that crash recovery is relatively likely to succeed, since software crashes are easy to detect. In that case, the crash-recovery risk P_{CR} is negligibly small as compared to bug-monitor risk P_{FN}, such that

$$P\{\text{failsafe trigger fault} \mid \text{verification fault}\} \approx P_{FN} , \quad (4)$$

and that, combining (2) and (4),

$$P_{LOI} = P_{FN} P_{VF} . \quad (5)$$

In subsequent analysis, (5) will be used to model the overall probability of a loss-of-integrity event causing system failure. Note that the above analysis (and the fault tree of Fig. 1) implicitly assume a vanishingly small probability that multiple unrelated bugs occur simultaneously, which is reasonable given a sufficiently thorough pre-service verification campaign (i.e., given that P_{VF} is sufficiently small).

BUG MONITOR IMPLEMENTATION

Before defining performance criteria for a bug monitor, it is helpful to first discuss how such a monitor would be implemented in practice. To this end, we briefly review details of a prototype bug monitor our team has recently implemented and tested.

General Description

Bug monitors we have implemented to date in [20] and [21] can be described in general terms by the block diagram of Fig. 2. The primary automation software and the bug-monitoring algorithm are two separate programs, each coded separately and linked at compile time. In this sense the design of the monitor is essentially independent from the design of the primary

automation, such that the source code for the automation software could conceptually remain proprietary and unknown to the designers of the bug monitor.

In the linking step, the bug monitor is configured to scan memory locations associated with certain variables used by the primary automation software. These *probes* allow the bug monitor to extract variable values associated with particular lines of the primary code. Probes may be analyzed directly, or they may be mapped via signal processing to a smaller set of derived values called *features*. A vector of probes and/or features is then compared to a *classification surface,* which represents a learned model of nominal behavior. If the feature vector lies inside the classification surface, the primary code is deemed to be operating normally, and so the primary code is allowed to continue controlling the aircraft. However, in the rare event that the feature vector lies outside the classification surface, the primary code is deemed to be operating anomalously, and control is switched to the failsafe kernel.

Bug Detector Implementation

In our recent work, we have instrumented Ardupilot, an open source autopilot [22] that has been configured to operate a wide range of autonomous and remotely operated platforms, from fixed-wing aircraft, to rotorcraft, submersibles, and ground vehicles. Ardupilot is a useful software testbed, in that it offers a well-documented bug database and version history. In fact, Ardupilot can easily be rolled back to re-introduce specific bugs from earlier versions. In working with Ardupilot, we have generated a monitor as a separate program, called *OScope*. We compile the two programs using Clang to obtain intermediate representations (IR), and link the two IR codes into a common executable using the Low-Level Virtual Machine (LLVM), an open source toolset originated by the University of Illinois [23].

In implementing our Oscope monitor, we considered four key design decisions:

- Probe selection: Which variables in the autopilot should be monitored?
- Time horizon: Should monitor classification consider only probe variables from a single time epoch (snapshot processing) or from multiple epochs (batch or recursive processing)?
- Feature selection: Should the set of sampled probe variables be subjected to signal processing (e.g. extraction of mean or dominant frequencies from a time signal) to generate a reduced set of features for classification, or should the probe values be used directly as classification features?
- Classification model: What model structure and training technique should be used to cluster features and distinguish between nominal and anomalous system behavior?

In implementing a proof-of-concept system, our guiding principal has been to keep our prototype as simple as possible in every respect in order to demonstrate feasibility; we envision optimizing performance at a later time, once we have shown the system works. With simplicity in mind, we addressed each of the bulleted design questions above to minimize prototype complexity.

First, we configured OScope to consider two basic methods for probe selection: (a) an input-output method, in which we selected 25 probe variables reading from sensors or writing to actuators, and (b) a randomized method, in which we selected 10 random variables from the set of all variables in Ardupilot. These methods of probe selection are unrelated to the location or nature of any particular bug, and so it is "fair" to test feasibility by applying these probes to detect known bugs from a bug database.

Second, we configured OScope's time horizon for snapshot analysis, with probe variables extracted at a single epoch and used for immediate bug-classification at that epoch. Though analysis of snapshot values (rather than a time-series) restricts the amount of data available and likely reduces our signal-to-noise ratio for classification, snapshot analysis offers a strong benefit. Notably, in snapshot analysis, physical-component dynamics cannot influence the time-evolution of the probe data. Thus, snapshot analysis isolates detected anomalies as *software* bugs, as distinguished from hardware-component faults that would otherwise correlate probe values in an anomalous manner over time.

Third, we configured Oscope to use the probe variables directly as classification features, with no additional signal processing. Although this design choice limits the signal-to-noise benefits of signal processing, avoiding signal processing simplifies implementation and interpretation. Moreover, absent filtering or other signal processing, the monitor responds immediately, such that time-to-alert is essentially zero.

Last, we configured Oscope to use one of the simplest machine-learning classifiers to model relationships among probe values. Specifically, we used an AdaBoost classifier, implemented as a set of decision trees. In addition to its ease of implementation, AdaBoost is advantageous in that its inner workings are relatively transparent and interpretable. Also, AdaBoost has proved useful in a number of automation-related studies, such as [24] and [25].

Preliminary Results

To test our prototype, we identified three relevant bugs from the Ardupilot bug database. Specifically, we considered bugs 2835 (large pitch actuation commanded when aircraft stalls during a turn), 6637 (aircraft flies away rather than entering a

circular holding pattern due to erroneous user-defined bank-angle parameter), and 7062 (failure to transition from climb to level flight for a VTOL aircraft). These bugs were all *difficult* bugs in the sense that they were not detected in early verification and validation of Ardupilot (indicating that the bugs might escape the pre-service verification block of Fig. 1). The bugs were also difficult in the sense that they did not cause the code to abruptly stop working (indicating they would have escaped the software crash recovery block of Fig. 1). These bugs escaping pre-service verification and software-crash recovery are ideal candidates for our proposed bug monitor, which is designed to serve as a last layer of defense.

For a basic feasibility evaluation, we generated a number of simulated flight trials using JSBSim, an open-source flight dynamics engine [26]. Approximately 20 trials were generated for each bug, each consisting of 3000 or more epochs individually classified as "nominal" or "faulted" by a human supervisor. In generating trials, we randomly perturbed initial conditions, waypoints, and disturbance forces. For this study, we simulated only one flight condition per bug; however it is worth noting that in our prior work (involving much simpler automation software), we were successful in training classifiers for diverse scenarios, in which the same model recognized both regular and balked landings as nominal (or correct) modes of operation of a control code, as long as no bug was active [20].

To keep our data analysis simple, we trained our classifier on half of the simulated trials and tested our classifier using the other half. The classifier was trained to a nominal alarm rate of 20%. Our results showed that the probe set consisting of input-output variables was indeed sensitive to the bug, with a median true-alarm rate of 65% for bug 2835, 55% for bug 6637, and 99% for bug 7062. Ten randomly selected probe sets (each consisting of 10 probe variables) were also considered and compared to the input-output variable set. The best random variable set for bug 2835 gave a true-alarm rate of 35%, for bug 6637 gave 58%, and for bug 7062 gave 98%. Analysis of confidence margins indicated all results were statistically significant relative to the nominal alarm rate of 20%.

These preliminary results demonstrate, for the first time, that detection of real bugs is possible in a real autopilot, even using a nonoptimized probe set. Clearly, some bugs are easier to detect than others, with the final bug (7062) being relatively easy to detect, with very few missed detections, and with the first bug (2835) being relatively hard to detect, with a great many missed detections. Perhaps surprisingly, the best random-variable probe sets performed nearly as well as the input-output variable set. This suggests that there is significant hidden structure within the code that might be exploited for optimal probe selection.

Although the performance metrics for our proof-of-concept are still far away from aerospace standards, we hypothesize that there is ample opportunity to enhance performance, given that we used the simplest possible strategies to build the monitor

including (a) randomized probe selection, (b) snapshot processing, (c) no signal processing on the probe values, and (d) rudimentary choice of machine-learning models for our classifier.

Unsupervised Bug Detection with a One-Class Model

As a practical consideration, it is important to acknowledge one additional design detail: the need for *unsupervised* machine learning. For the bug-detection application, we must assume bugs can take nearly any form, and we cannot expect that new bugs will resemble bugs observed previously in the bug database. In this sense, it is not particularly useful to train the bug detection model with *supervised* learning methods, which rely on a human expert to label when known bugs are expressed.

A more useful approach for designing a monitor is to use an unsupervised approach, which does not rely on any model of a bug. To this end, we consider a one-class model, in which the model is trained only to recognize nominal behavior, such that any observation inconsistent with nominal behavior must be considered to be an anomaly. This approach is unsupervised in the sense that no labeling is required for the training data set.

As long as the training data are constrained to represent predominantly nominal behaviors (allowing for a few rare bug conditions mixed in), then the one-class model can be trained without explicit supervision. This is an important detail, and it bears restating. As long as bugs are relatively rare in the training data, a one-class model can be defined using unlabeled data. Bugs in the training data will inadvertently influence the classification surface that represents bug-free operation; however, this influence will be small as long as bugs are sufficiently rare. The analysis that follows will explicitly model this effect, by using the probability that bugs escaping pre-service verification as a model of their frequency of appearing in the nominal training data. A one-class model identifies feature values as lying inside or outside of the nominal classification region. Fig. 4 illustrates the conceptual difference between the supervised two-class model described for our proof-of-concept Ardupilot implementation and the unsupervised one-class model we intend to use in future work. The figure is purely for illustrative purposes, with all points generated by a random number generator (and not from the Ardupilot experiments). The figure models a situation in which a single bug triggers intermittently. Bug-free instances are shown as closed circles and instances where the bug is active are shown as open circles. Note that the feature vector changes at each epoch due to variations in software variables that are unrelated to the bug. As such, there is a distribution of sampled feature values for both the buggy and bug-free cases and that these distributions may be different in structure in the general case, as shown in Fig. 4.

As shown on the top of the figure, the two-class model divides the feature space into two regions (shaded and unshaded) to separate data vectors that have been labeled by a human expert. In the case shown, each point represents a feature vector

associated with monitor probe values sampled at one instant in time. Illustrated feature vectors are only two-dimensional, with the feature-A dimension plotted on the horizontal axis and feature-B dimension on the vertical. Feature vectors labeled as nominal are shown as closed dots; feature vectors labeled as anomalous are shown as open circles. The boundary between the two regions is identifying by a machine learning algorithm (such as Adaboost), so as to capture most of the closed dots (nominal points) in one region and most of the open circles (anomalous points) in another region. In this case, the boundary between the two classification regions is a line at the interface of the shaded and unshaded regions. When points fall in the shaded region they are classified as nominal and in the unshaded region they are classified as anomalous. Note in the figure that a few stray dots are misclassified, including some closed dots (nominal points) that fall in the unshaded (anomalous) classification region, which are false alarms, and some open circles (anomalous points) that fall in the shaded (nominal) classification region, which are missed detections. A "good" classification surface is one that minimizes both the number of false alarms and the number of missed detections.

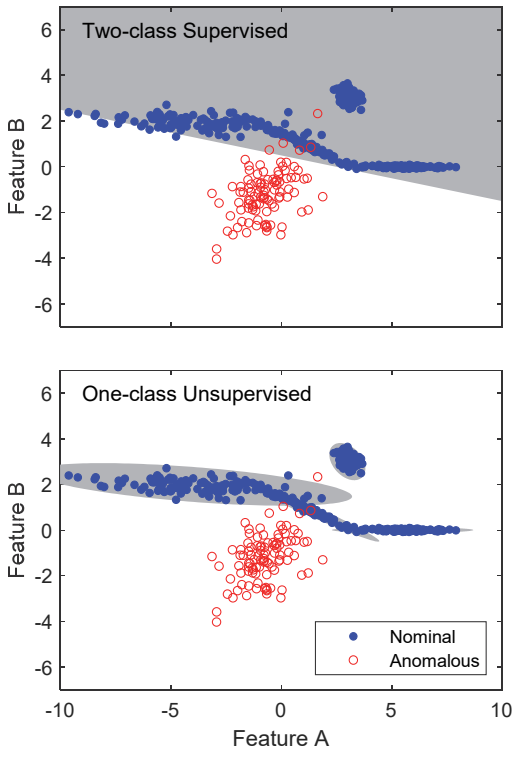

Fig. 4. Classification surfaces for two-class supervised (top) and one-class unsupervised (bottom) models. Horizontal and vertical axes represent the two values from a two-element feature vector.

As shown on the bottom of the figure, an alternative classification strategy is to build a tight bound around the nominal points. In the illustration, the distribution of the nominal points is irregular; as such a clustering algorithm (such as Expectation Maximization) can be used to define multiple clusters within the nominal data. The union of these clusters defines the region of the feature space associated with nominal behavior. In this case, four Gaussian-like clusters are identified, and an elliptical (shaded) region is associated with each of the four nominal clusters. Three of these ellipses overlap, but the fourth is not contiguous. The union of the four ellipses, defines the *nominal* classification region. As desired, the majority of the closed dot (nominal) points lie in the nominal classification region, while the majority of the open circle (anomalous) points lie outside. Because only nominal behaviors are modeled, this type of one-class model is agnostic to the nature of unknown bugs. As such, the one-class model should allow detection of any type of bug that causes the software to behave in an off-nominal fashion, whether or not the new bug is similar in form to a bug previously observed during software verification.

Of course, operating in unusual environmental conditions could also result in an anomaly that might trip the monitor, even in the absence of a bug. In such cases, it is nonetheless prudent and conservative to seek safety (e.g. land) since the monitor alert could just as easily be the result of a bug. By examining logged cases, we envision that it will be possible to determine whether anomaly events are caused by a bug or a new operating condition and to either fix the bug or to update the trained one-class model accordingly.

In order to construct the one-class model for the *nominal* region, as shown on the bottom of Fig. 4, it is critical to collect the right training data. In particular, the training data must (1) capture the full range of expected operational conditions and (2) ensure that most data points are unaffected by a bug. The first above requirement will be satisfied by pre-service verification. In other words, the pre-service verification study should run through all the most likely operational conditions expected in the field. Thus, the pre-service verification study that exists today to verify current-generation aviation software will take on a new dimension, as its secondary purpose will be to generate a training data set for the one-class bug monitor. The second above requirement is also satisfied by pre-service verification. Because pre-service verification ensures that bugs are rare, very few bug-affected values will appear into the training set. In concept, any bug-affected values that do pass through verification will be close enough to nominal that they will satisfy verification requirements, and so the appearance of a few bugs in the training set should not significantly corrupt the training of the classifier. A quantitative analysis (see the next section) confirms that allowing a few buggy values in the training data does not significantly corrupt the classification surface.

PERFORMANCE CRITERIA FOR BUG MONITOR

This section defines and quantifies relevant performance criteria for an online bug monitor. As discussed in the prior section, our prototype monitor proves the bug-detection concept, but does not yet meet performance levels suitable for practical deployment in an aviation system. The question remains: what monitor performance levels are required to provide practical benefit in relaxing pre-service verification? To address the question, this section introduces a simplified analytical model that enables exploration of the bug-monitor design space.

Performance Criteria Definitions

For a bug monitor to be effective, it must reliably and quickly detect bugs large enough to result in a potential system fault, with very few false alarms. This qualitative statement can be decomposed into four quantitative performance criteria: time-to-alert, loss-of-integrity probability, minimum-detectable bias, and alarm probability. To characterize monitor performance, we define four monitor performance criteria as follows.

Time-to-alert (*TTA*): This is the allowable time between the moment a bug-induced fault becomes hazardous and the moment at which the failsafe feature is activated. If the monitor fails to trigger the failsafe within the *TTA*, then the event is considered to be a false negative (or *missed detection*) event. For a safety-critical aviation application, *TTA* may be as short as 2 seconds [27].

Loss-of-integrity probability (P_{LOI}): This is the allowable probability that a bug results in a major failure without the monitor triggering an alert within the *TTA*. The term *major failure* here means an event that causes loss of life or substantial loss of property. For the proposed architecture, P_{LOI} is quantified by (5). Recall that integrity probabilities are usually quantified as risks per unit time (e.g. risk of a hazardous fault due to a bug per hour), but that temporal effects are not explicitly addressed in this paper.

Minimum-detectable bias (b): This is the smallest bias that can be reliably detected and for which the P_{LOI} requirement can be satisfied. Bug-induced faults that are not clearly observable to be anomalies (e.g. those which are not significantly outside the monitor's classification surface) will not meet the P_{LOI} requirement. In other words, bug-induced for which the monitor vector remains small are unobservable to the monitor; hence, the software must be constructed to be robust to such faults. The minimum detectable bias b sets the robustness criterion. Any fault that, on average, corrupts software values by a magnitude less than b must not result in a major failure.

Alarm probability (P_A): This is the allowable probability that the monitor engages the failsafe per unit time, considering both true positives and false positives (aka, *true detections* and *false alarms*). Whether the failsafe kernel is triggered by a hazardous bug or by an incorrect detection, the ensuing emergency response is an extreme inconvenience. Because such alarm events suspend use of the primary automation software, they might be labeled *continuity breaks*, a term frequently used to describe alarm events in other aviation applications [28].

Quantifying Monitor Performance

To support design-space exploration, it is now useful to introduce equations to quantify the above performance criteria using a probabilistic model. The Ardupilot infrastructure described above is computationally intensive to run, so statistical exploration of the design space is impractical.

In particular, this section introduces probabilistic expressions for the alarm probability P_A and the minimum-detectable bias b. A probabilistic expression for the loss-of-integrity probability P_{LOI} was already developed as (5). For general monitor designs, our future work will eventually need to address timing issues, such as discussion of the time basis for probabilities, time correlation, and *TTA*. However, *TTA* is not an immediate concern for snapshot monitors like the Ardupilot monitor described above, since snapshot monitors respond immediately with a *TTA* of zero.

Alarm Probability (P_A): The alarm probability combines both true and false alarms. Thus

$$\begin{aligned} P_A = {} & P\{\text{true positive} \mid \text{hazardous bug}\}\, P\{\text{hazardous bug}\} + \\ & P\{\text{false positive} \mid \text{no hazardous bug}\}\, P\{\text{no hazardous bug}\} \end{aligned} . \tag{6}$$

The probability of a bug being present is equal to the probability of a verification fault P_{VF}. Conservatively modeling as hazardous all bugs large enough to be detected, $P\{\text{hazardous bug}\}$ is upper bounded by P_{VF}. Note that P_{VF} lumps together the two mechanisms for detecting bugs (bug monitoring and crash recovery). Also note that the probability of a true-positive approaches one for both mechanisms and that, with very mild conservatism, the probability $P\{\text{true positive} \mid \text{hazardous bug}\}$ has been rounded up to one. Thus,

$$P_A \approx P_{VF} + P\{\text{false positive} \mid \text{no hazardous bug}\}\, P\{\text{no hazardous bug}\} . \tag{7}$$

Equation (7) can further be simplified by recognizing that the probability of no hazardous bug is the complement to the probability of a hazardous bug. In other words, $P\{\text{no hazardous bug}\} = 1 - P_{VF}$. The label P_{FP} will be used to identify the probability of a false positive, meaning the probability that the failsafe is triggered given that no bug is present. Thus,

$$P_A = P_{VF} + P_{FP}\left(1 - P_{VF}\right). \tag{8}$$

The classifier acts on the monitor vector, a vector of probes and/or features that we will label $\boldsymbol{x} \in \mathbb{R}^N$. Based on training of a one-class classifier, a subspace $\Omega \subset \mathbb{R}^N$ will be defined as representing nominal monitor behavior $\left(\mathbf{x} \in \Omega \rightarrow \text{nominal}\right)$. This implies that any vector **x** in the complementary subspace will be classified as an anomaly $\left(\mathbf{x} \in \Omega^C \rightarrow \text{anomaly}\right)$, which will in turn trigger the failsafe. The size of classification region can be scaled to provide a desired level of confidence in classification. If the nominal subspace Ω is made larger, for instance, the probability of a false positive is reduced. A quantitative statement of the false-positive probability P_{FP} is

$$P_{FP} = 1 - \int_{\Omega} p_{nom}\left(\mathbf{x}\right) d\mathbf{x}. \tag{9}$$

Here the probability distribution $p_{nom}\left(\mathbf{x}\right)$ is the probability density function for the monitor vector when no bug is present. Note that in the machine learning community, the integral on the right-side of the above equation is often called *specificity*. (That is, specificity is $1 - P_{FP}$).

An important wrinkle for practical implementation is that the classification surface will be trained based on unlabeled data that may actually contain bugs. This type of unsupervised learning is required by the nature of the problem. In other words, if a bug could be labeled, then it would be removed prior to training and deployment; however, the *raison d'etre* for the monitor is to identify bugs that persist into deployment and that must therefore be present during training.

Fortunately, if training data are collected during a pre-service verification campaign, then it is at least known that the probability of bugs is rare (less than or equal to P_{VF}). Thus, even though latent bugs are present in the training data, they have little impact on the definition of the classification surface, as long as the bug probability P_{VF} is somewhat smaller than the target false-positive probability P_{FP}.

An important distinction between the false-positive rate P_{FP} and the total alarm rate P_A, which considers both false and true positives, is that the designer can only directly measure P_A. In other words, during training, samples cannot automatically be classified as "buggy" or "bug-free," and so alarms cannot be categorized as true or false positives. It is still possible to discuss a false-positive rate in theoretical terms, as in (9), and is indeed useful to quantify the false-positive probability this way for analytical purposes, such as in simulation-based design-space exploration.

*Minimum Detectable Bias (*b*):* Recall that the minimum detectable bias is the smallest anomaly that can reliably be detected via classification.

In order to define this bias b, we must first model another aspect of the classification process. In particular, we must model the false-negative (or missed detection) probability P_{FN}, which describes the probability that the monitor fails to alert when an observable bug is present. Defining $\boldsymbol{\mu}$ to be the mean value of the monitor vector for a given bug and $p_{bug}(\mathbf{x};\boldsymbol{\mu})$ to be the probability density function (PDF) describing values of the monitor vector when that bug is present, the false-negative probability for that bug can be obtained by integrating over the nominal subspace Ω:

$$P_{FN} = \int_{\Omega} p_{bug}(\mathbf{x};\boldsymbol{\mu})\, d\mathbf{x} \,. \tag{10}$$

Within the machine-learning community, the term *specificity* is commonly used to refer to the complement of the false-negative probability. (In other words, specificity is 1- P_{FN}).

Although it may not be possible to know the distribution $p_{bug}(\mathbf{x};\boldsymbol{\mu})$ for any particular bug, it may be possible to develop a conservative overbounding model to describe the distribution. Overbounding is a common procedure used in aviation applications to conservatively model distributions for unknown fault modes [29]-[32]. As a starting point, it can be useful to assume that the distribution of the features is similar whether or not the bug is present, and that the bug simply acts to shift the mean of the feature-vector distribution. Given that the structure of the buggy and bug-free distributions may be different in the general case (as illustrated in Fig. 4), this assumption that the bug only shifts the distribution mean without otherwise affecting distribution shape is clearly an approximation, one that should be revisited in future work. Similar assumptions are commonly made in analysis of aviation navigation systems [33]-[35], and these assumptions can be rigorously justified if there exists an overbounding noise distribution, meaning a single model that conservatively describes various bugs as well as nominal operation. In any case, we assume here that an overbound for all bugs can be defined, and use this reasonable assumption as a basis for

preliminary characterization of monitor performance. Assuming an overbounding distribution exists to model the shape of the nominal and buggy distributions, then the key feature of the overbounding distribution is simply its mean $\boldsymbol{\mu}$.

Let us assume that bugs can shift the mean of the overbounding distribution away from the nominal mean $\boldsymbol{\mu}_0$, and that the value of the buggy mean $\boldsymbol{\mu}$ is arbitrary. In this case, it is useful to determine which values of the unknown mean can be detected reliably and which values are too small to detect (meaning that the code must be hardened to be robust to these small biases). To determine this minimum detectable value, the first step is to invert equation (10) to find the bias that corresponds to a worst-case specified P_{FN}. In general, there will be multiple solutions to this inverse problem. Define the set of solutions to be U.

$$U = \left\{ \boldsymbol{\mu} \mid P_{FN} = \int_{\Omega} p_{bug}(\boldsymbol{x}; \boldsymbol{\mu}) d\boldsymbol{x} \right\} \tag{11}$$

The minimum detectable bias b is the smallest distance between any $\boldsymbol{\mu} \in U$ and the nominal mean $\boldsymbol{\mu}_0$, which describes the center of the nominal (bug-free) feature distribution.

$$b = \min_{\boldsymbol{\mu} \in U} \left(\|\boldsymbol{\mu} - \boldsymbol{\mu_0}\| \right) \tag{12}$$

This expression can be computed analytically, even when the mean of the overbounding model is unknown. Note that the feature-value distribution factors into b through the integral for the false-negative rate P_{FN}, and that the overbound on the random distribution of feature values is needed to conservatively evaluate the integral. The result is sensitive to the difference between the bug's unknown mean $\boldsymbol{\mu}$ and a reference location $\boldsymbol{\mu_0}$, a variable which describes the center of any of the clusters unioned to define the classification region, as visualized in Fig. 4.

As defined by (12), the minimum detectable bias is a measure of how small a bias can reliably be detected. Any bias with a monitor vector whose magnitude is smaller than b cannot be reliably detected, meaning that the missed-detection probability is greater than P_{FN} (and potentially approaching one). Since bugs with magnitude smaller than b are difficult to detect, the primary control software must be designed to handle corrupted data when a small-magnitude error (magnitude less than b) is present. In this sense, the value b also defines the maximum bug magnitude which the software must be designed to tolerate.

Though the value of P_{FN} cannot be measured directly, it can be related to the specification for P_{LOI} using (5).

SIMULATED PERFORMANCE ANALYSIS

In a broad sense, the key performance metrics P_{LOI}, P_A, and b are related to each other and to the intermediate variables P_{FN}, P_{FP}, P_{VF}, and T through (5), (8), and (12). These relationships can be simulated to explore the space of possible monitor designs if the PDFs for the buggy and bug-free cases can be modeled.

PDF Models

For the purposes of roughly characterizing the monitor design space, this paper models the nominal monitor vector PDF as a one dimensional Gaussian distribution with zero mean and unit variance.

$$p_{nom}(x) = p_g(x;0,1) \tag{13}$$

Here the function p_g describes the Gaussian PDF for a monitor value x, parameterized by a mean of zero and a variance of one. The overbounding variance can be set to one without loss of generality simply by normalizing the monitor statistic appropriately. The bigger approximations here are treating the monitor vector as a one-dimensional random variable and more specifically as a Gaussian-distributed variable. In concept it is possible to model the monitor-vector distribution as a multivariate random vector with an arbitrary PDF; however, as illustrated in Fig. 4, even a complicated feature distribution can potentially be modeled as a set of multi-dimensional Gaussian distributions. Building on this concept that a Gaussian distribution is reasonably representative of many nominal training sets, we simplify our analysis to aid interpretation of our results and introduce a one-dimensional Gaussian distribution as a model of the feature-vector distribution for the bug monitor. In short, the assumption of a one-dimensional Gaussian PDF is a reasonable starting point, which simplifies analysis greatly and which reflects a level of abstraction appropriate for design-space exploration.

As for the anomaly-case PDF distribution, this PDF is also modeled here as a Gaussian PDF. Specifically, the PDF $p_{bug}(x)$ is biased by a mean value of μ, reflecting the level to which the bug corrupts variable values, and the standard deviation is defined as σ_{bug}. With these assumptions, the bug-case PDF is

$$p_{bug}(x) = p_g\left(x;\mu,\sigma_{bug}^2\right). \tag{14}$$

The proposed bug-corrupted PDF is intended to be an overbound in the sense that it models a generalized distribution that is as conservative as any real distribution (with wider tails that ensure the false-negative probabilities will be evaluated to be at least

as large as for the true PDF). In fact, we will assume that both the bug-case and bug-free PDFs are overbounds and that their standard deviations are scaled to be the same ($\sigma_{bug} = 1$). This assumption of an identical overbounding variance for the faulted and fault-free distribution is common in analyses of other aviation-safety applications, for example in [27] and [36].

For a one-dimensional PDF model, the classification region Ω can be identified by a pair of thresholds, one above and one below zero. Since our nominal PDF (13) is symmetric about zero, it makes sense to assign thresholds symmetrically at the values $x = \pm T$. Such thresholds are illustrated in Fig. 5. The thresholds are shown as bracketing the nominal distribution, as in Fig. 5 (left), but only capturing one tail of the bug-case distribution, as in Fig. 5 (right).

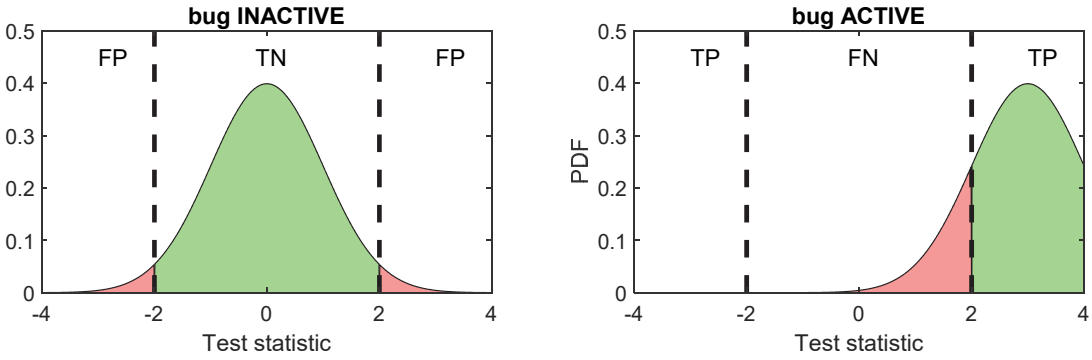

Fig. 5. Model of feature distribution

For a symmetric threshold pair and a Gaussian nominal distribution, equation (9) for P_{FP} can be rewritten as

$$P_{FP} = 1 - \int_{-T}^{T} p_g\left(\mathbf{x}; 0, 1\right) d\mathbf{x} \,. \tag{15}$$

This equation can also be inverted to express the threshold T in terms of the false-positive probability P_{FP}. If P_g^{-1} is the inverse of the probability integral above, then

$$T = P_g^{-1}\left(1 - P_{FP}\right). \tag{16}$$

When using numerical analysis packages, the inverse density function of a 1D Gaussian integrated between thresholds can be evaluated using a chi-square cumulative density function with one degree of freedom. Using Matlab for instance, where the inverse chi-square cumulative density function is labeled `chi2inv`, the threshold T is

$$T = \sqrt{\texttt{chi2inv}\left(1 - P_{FP}; 1\right)}. \tag{17}$$

Likewise, the minimum-detectable bias b can also be rewritten in terms of the Guassian bug-case model by combining (12) and (14). Without loss of generality, $\boldsymbol{\mu_0}$ is assumed zero for this analysis. Thus, the minimum detectable bias can be obtained by finding the value of b that solves a constraint equation:

$$b = \underset{b>0}{\text{solve}}\left[P_{FN} - \int_{-T}^{T} p_g\left(x; b, 1\right) dx = 0 \right]. \tag{18}$$

This type of solution can be obtained readily with a numerical package like Matlab (e.g using Matlab's `fsolve` command). Note that the solution to the above equation is not unique, since the integral is symmetric for positive and negative values of b. A unique solution can be obtained, however, if b is restricted to be positive.

The full design space can now be characterized by combining the general equations (5) and (8) with the model-specific equations (17) and (18). These equations relate seven parameters with four equations. For our purposes, it makes sense to define four of the variables as dependent parameters: $\left\{P_{FN}, P_{FP}, P_{vf}, T\right\}$. The remaining variables, which are the independent parameters, are the three key performance metrics: $\{P_{LOI}, P_A, b\}$. In other words, if specifications are given for the three key performance metrics, then the monitor design is completely specified by the four equations (5), (8), (17), and (18). In this sense, the surface of possible designs might be viewed as a vector function **f** of the three key performance metrics:

$$\begin{bmatrix} P_{FN} \\ P_{FP} \\ P_{vf} \\ T \end{bmatrix} = \mathbf{f}\left(P_{LOI}, P_A, b\right). \tag{19}$$

Design-Space Dimension Reduction

In fact, a numerical analysis reveals that this three-dimensional design space acts more like a function of only two variables rather than three. In this section we show that it is reasonable to introduce a heuristic fifth equation into the design

characterization, in order to approximate (19) as a vector function **g** of only two independent variables. This function might be written as

$$\begin{bmatrix} P_{FN} \\ P_{FP} \\ P_{vf} \\ T \\ b \end{bmatrix} = \mathbf{g}\left(P_{LOI}, P_A\right). \tag{20}$$

In order to explain this dimension reduction, we note that the value of the verification-fault probability P_{VF} must by definition be smaller than the alarm probability P_A, as discussed above and as evident from (8). Given this constraint, it is useful to introduce a variable $\phi \in [0,1]$ to describe the ratio of the two probabilities:

$$\phi = \frac{P_{vf}}{P_A}. \tag{21}$$

A different behavior is evident at each extreme of the values of ϕ. At the top end, for instance, where $\phi \to 1$, all of the alarm probability is dedicated to true alarms, meaning that the bug monitor must be set inactive. In other words, near the limit $\phi \to 1$, $P_{vf} \to P_A$ according to (21), and $P_{FP} \to 0$, according to (8). Furthermore, $T \to \infty$ as $P_{FP} \to 0$, by inspection of (15). Setting the monitor threshold to infinity is essentially equivalent to turning off the monitor, as the monitor will never alarm. By contrast, at the bottom end of the range, where $\phi \to 0$, the pre-service verification becomes perfect. In quantitative terms, $P_{vf} \to 0$, according to (21), as $\phi \to 0$. In this limit, the monitor is also inactive, in this case because no bugs persist after the pre-service verification and so no monitor is needed.

Given that the monitor is inactive at both extremes of $\phi \in [0,1]$, we need only consider values in the middle of the range. As it turns out, model performance is relatively flat for values of ϕ in the middle of its range, and so as a heuristic, it is reasonable to introduce a constraint equation that sets ϕ to a constant value.

To see that performance is only weakly affected by changes to ϕ, consider Fig. 6. The figure illustrates threshold T as a function of ϕ for widely varying combinations of P_{LOI} and P_A. As expected, the threshold diverges in all cases near the upper limit, as

$\phi \to 1$. Otherwise, the threshold is relative flat, with a value very similar to the lower-case bound, where all of the alarm probability is given to false positives, where T is obtained by setting $P_{FP} = P_A$ in (17).

For the models used in this paper, a reasonable value of the heuristic constraint is to set $\phi = 0.2$. At this operating point, the pre-service verification fault probability is relatively large (20% of the total alarm budget) and the threshold is relatively close to its lower bound, which is good in the sense that a lower threshold makes for a more sensitive monitor. Realistically, any ratio could be selected in the middle range of $\phi \in [0.1, 0.6]$, with relatively minimal tradeoffs between making the pre-service verification more stringent (decreasing P_{vf} as ϕ decreases) and making the monitor less sensitive (increasing T as ϕ increases).

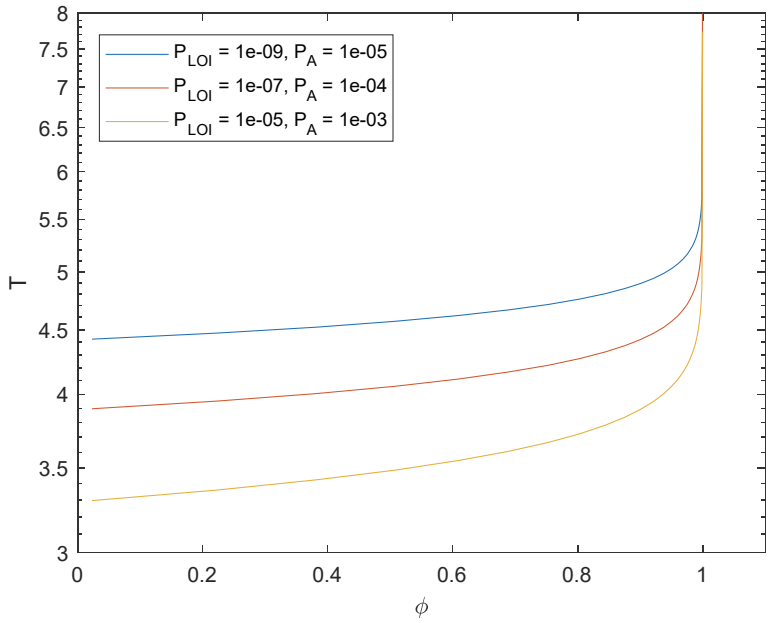

Fig. 6. Alert probability PA and verification-fault probability PVF are related by the parameter ϕ

Whatever value of ϕ is selected as the nominal operating point, the new constraint equation can be derived by manipulating (21) to give:

$$P_{vf} = \phi P_A. \tag{22}$$

This equation can be combined with other design equations. For example, substituting (22) into (8) gives

$$P_{FP} = \frac{(1-\phi)P_A}{1-\phi P_A}. \tag{23}$$

Also, substituting (22) into (5) gives

$$P_{FN} = \frac{P_{LOI}}{\phi P_A}. \tag{24}$$

Importantly, combining (24) with (18) creates a constraint equation that directly relates the key performance metrics P_{LOI}, P_A, b, implying that only two of the key performance metrics can be treated as independent parameters once ϕ is set to a constant.

Characterizing Design Space

As a final analysis step, it is instructive to visualize the design surface **g**, as defined by (20), over a representative range of the independent design parameters P_A and P_{LOI}. Each of the remaining parameters in the set $\{ b, P_{FN}, P_{FP}, P_{vf}, T \}$ can be studied over this range.

First, consider the relationship of the minimum-detectable bias b to the independent parameters P_A and P_{LOI}. This relationship is represented as a contour plot in Fig. 7. Contours of the bias b are expressed as multiples of the standard deviation, shown increasing from 2 to 8 downward, as P_{LOI} shrinks. This trend is not unexpected; in order to enhance integrity (achieve lower P_{LOI}) requires fewer missed-detection risk by the online bug monitor, and so larger margins of robustness are needed as achieved by increasing b.

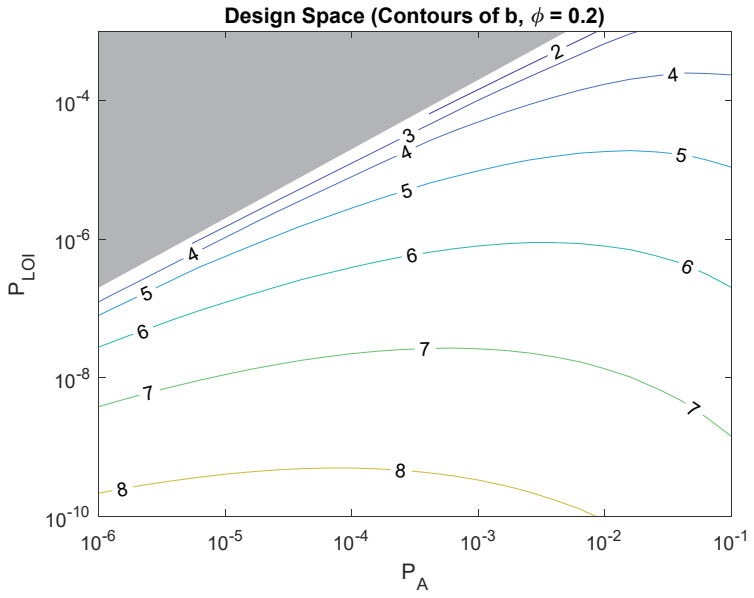

Fig. 7. Design space for minimum-detectable bias b

In observing Fig. 7, it is also notable that b does not increase monotonically when moving from left to right, through increasing P_A. The reason the peak value of b lies in the middle of the range of P_A is two-fold. First consider moving to the left on in Fig. 7. In this direction $P_{LOI} \to \phi P_A$, which implies that the monitor is ineffective (e.g. $P_{md} \to 1$) and that all bug mitigation is the responsibility of pre-service verification. Thus, moving to the left side of the plot, values of b decrease sharply since the monitor is essentially inactive. In fact, values of $P_{LOI} \geq \phi P_A$ cannot even be achieved, and so this region of the design space is shaded gray to indicate that it is infeasible. Second consider moving to the right on in Fig. 7. In this direction, P_A increases implying for fixed ϕ that the tolerance for false alarms also increases (P_{FP} increases) and hence that the monitor can be more sensitive (reduced T and b).

Design parameters other than b can also be investigated, but it turns out that trends are very simple. For instance, contours of threshold T are shown in Fig. 8. For constant ϕ, the threshold is only a function of the alarm probability P_A, and so all of the contours appear as vertical lines. Similarly, contours of P_{vf} (not shown) are also vertical lines.

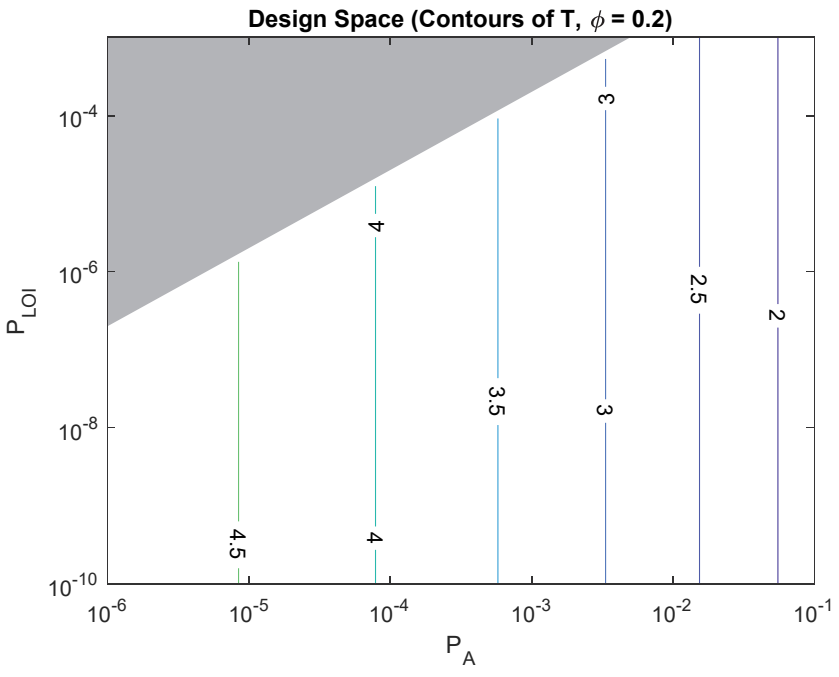

Fig. 8. Design space for minimum-detectable bias T

Design Implications

Certain general design recommendations can be inferred from our analysis of the monitor design space. Most notably, it is important for a practical design to keep P_{LOI} low, to assure a level of integrity sufficient for safety-critical operation, and to keep P_A low, to preserve continuity, since alarms cause regular operations to end as the failsafe algorithms become active. In other words, it is generally desirable to operate toward the lower left region of Fig. 7 and Fig. 8. Moreover, it is important to operate some distance from the infeasible region, shown gray in both figures, since the monitor becomes ineffective approaching the boundary of this region, as discussed above. In short, operating somewhere in the lower-left (or southwest) quadrant of the figures is most desirable.

The major limitations on preceding arbitrarily far into the lower-left quadrant of the figures is that the robustness requirements become large moving downward (b increases as P_{LOI} decreases) and the pre-service verification requirements become stringent moving to the left (P_{VF} decreases as P_A decreases). The robustness issue is significant because the code must be configured to work with increasingly large undetected faults (large b) in order to achieve enhanced integrity (lower P_{LOI}). The pre-service verification issue is significant because the whole point of introducing the monitor is to reduce the severity of pre-service verification requirements (for fixed level of integrity P_{LOI}) in order to promote faster system development. Recall the earlier motivating question, which asked if the levels of integrity proposed in Fig. 1 were feasible for a practical implementation. The answer is that such a design is conceptually possible. By setting the alarm probability to a value of P_A equal to $5 \cdot 10^{-4}$, and by targeting a ϕ ratio of 0.2, the pre-service verification requirement would become $P_{vf} = 10^{-5}$. It would still be possible to achieve an overall integrity risk of $P_{LOI} = 10^{-9}$ (as shown in Fig. 1) if b were set to a value of approximately 8 times the standard deviation of the monitor statistic, at least assuming the simple Gaussian model used in this paper.

The major tradeoff is the robustness requirement implied by b. Because the monitor's sensitivity is limited in detecting small bugs, the code must be made robust enough to tolerate corrupted-data issues caused by small bugs. For perspective, the code will experience variations in its variables that amount to a three-standard-deviation value of the monitor statistic with a probability of about 10^{-3} (for a Gaussian PDF). In other words, the three-sigma level might be considered to be a rare-nominal event in the absence of a bug. By comparison, the code must be made to be robust to anomalies at the eight-sigma level, nearly three times the size of this rare-nominal event. This tradeoff seems quite modest given the enormous, five-order-of-magnitude reduction in the requirements for pre-service verification.

Implications for Monitor Prototype

Our design-space exploration suggests that the bug-monitoring architecture described in Fig. 1 may in fact be realizable. To reduce the requirements on pre-service verification by five orders of magnitude, from $1 \cdot 10^{-9}$ to $2 \cdot 10^{-4}$, would require a significant enhancement in the performance of our current prototype. Specifically, an alarm rate of $5 \cdot 10^{-4}$ per unit time would be required, which is a significant improvement over the performance of our prototype monitor, which featured an alarm rate of $3 \cdot 10^{-1}$ per epoch. However, given the simplicity of our implementation and the fact that many avenues exist to enhance monitor performance – through introduction of careful probe selection, an extended time horizon, signal processing, and enhanced machine-learning models – an alarm rate of $5 \cdot 10^{-4}$ per unit time may, in fact, be achievable. This hypothesis suggests a rich vein of research to explore in our future work.

Future work should also revisit analytical assumptions made in this paper. For instance, a simple Gaussian model was assumed for our design-space exploration. It was further assumed that an overbound could be defined for the feature vector probability density function. Before leveraging such assumptions to make a safety case, the assumptions should be rigorously tested against representative data from bug logs for deployed aviation software. Furthermore, new research is needed to show whether aviation software can indeed be made robust to small bugs (i.e. bugs for which the monitor static falls below a minimum detectable level).

It is worth noting that the most likely applications of our proposed work are low-cost unmanned aircraft applications, where safety must be assured for operation in the airspace, but where cost drivers prohibit intensive pre-service verification to the standards expected for today's large commercial aircraft. Though an alarm rate of $5 \cdot 10^{-4}$ per an appropriate unit time (say, per hour) would be prohibitive for a manned aircraft, such an alarm rate is acceptable for a great many potential applications of automated drone aircraft.

CONCLUSION

This paper presents a novel concept for relaxing pre-service software verification requirements for safety-critical aviation automation by introducing online bug-monitoring. The concept would provide significant benefit for emerging autonomous systems, in which software complexity is rapidly increasing. The analysis presented in this paper introduces a safety case for the combination of pre-service verification and bug monitoring. Also, the paper identifies key performance metrics to evaluate the bug monitor. A preliminary analysis based on a simple (Gaussian) model of the variability in the monitor statistic suggests that significant safety benefits can be achieved with reasonable tradeoffs; for instance, an example is discussed in which integrity is

preserved while relaxing pre-service verification requirements by five orders of magnitude. The tradeoff is a relatively modest increase in robustness, such that the code can tolerate small undetectable bugs, about three times larger than rare-normal variations in the values of software variables. In short, bug monitoring offers great potential benefits to shorten the software development cycle for aviation systems without sacrificing overall system safety.

ACKNOWLEDGEMENTS

The author gratefully acknowledges the National Science Foundation for supporting this research through grants 1329341 and 1836942.

REFERENCES

[1] Enge, P., T. Walter, S. Pullen, C. Kee, Y.C. Chao, and Y.J. Tsai, "Wide area augmentation of the global positioning system," *Proceedings of the IEEE* 84(8):1063-1088, 1996.

[2] P. Enge., "Local area augmentation of GPS for precision approach of aircraft," *Proceedings of the IEEE*, 87(1):111-132, 1999.

[3] Hegarty, C. and Chatre, E., "Evolution of the global navigation satellite system (GNSS)," *Proceedings of the IEEE* 96(12):1902-1917, 2008.

[4] Rife, J. and Pullen, S., "Aviation applications," *GNSS Applications and Methods*, S. Gleason and D. Gebre-Egziabher, Eds. Artech House, Norwood, MA: Artech House, 2009, pp. 245-268.

[5] Dvorak, Daniel L. *NASA Study on Flight Software Complexity*. NASA office of chief engineer (2009). https://www.nasa.gov/sites/default/files/418878main_FSWC_Final_Report.pdf

[6] Augustine, Norman R. *Augustine's laws*. AIAA, 1997.

[7] Knight, John C. "Software challenges in aviation systems." *International Conference on Computer Safety, Reliability, and Security*. Springer Berlin Heidelberg, 2002.

[8] Rushby, John. "New challenges in certification for aircraft software." *Proceedings of the ninth ACM international conference on Embedded software*. ACM, 2011.

[9] Pasztor, A., Tangel, A., Wall, R., and A. Slider. "How Boeing's 737 MAX Failed," *The Wall Street Journal*, March 27, 2019. https://www.wsj.com/articles/how-boeings-737-max-failed-11553699239

[10] Butler, Ricky W., and Caesar A. Munoz. "Formally verified practical algorithms for recovery from loss of separation," Technical Memorandum, NASA TM-2009-215726, 2009.

[11] H. Herencia-Zapana, J.-B. Jeannin, and C. Muñoz. "Formal verification of safety buffers for state-based conflict detection and resolution," Proc. *International Congress of the Aeronautical Sciences*, 2010.

[12] Kim, Moonjoo, et al. "Formally specified monitoring of temporal properties." Real-Time Systems, 1999. *Proceedings of the 11th Euromicro Conference on. IEEE*, 1999.

[13] J. Rushby, "Formal Methods and the Certification of Critical Systems," Computer Science Laboratory, SRI International, Menlo Park, CA, SRI-CSL-93-7, Dec. 1993.

[14] Thomke, Stefan, and Donald Reinertsen. "Agile product development: Managing development flexibility in uncertain environments." *California management review* 41(1):8-30, 1977.

[15] Greer, Des, and Yann Hamon. "Agile software development." *Software: Practice and Experience* 41(9):943-944, 2011.

[16] Dybå, Tore, and Torgeir Dingsøyr. "Empirical studies of agile software development: A systematic review." *Information and software technology* 50(9):833-859, 2008.

[17] RTCA. *DO-178C: Software Considerations in Airborne System and Equipment Certification*. RTCA, December 13, 2011.

[18] Rierson, Leanna. Developing safety-critical software: a practical guide for aviation software and DO-178C compliance. CRC Press, 2017.

[19] Gray, Jim. Why do computers stop and what can be done about it? *Symposium on reliability in distributed software and database systems*. 1986.

[20] Huang, Hu, Samuel Guyer, and Jason Rife. "Applying machine learning for run-time bug detection in aviation software." *AIAA Infotech@ Aerospace*, 2016.

[21] Huang, Hu. *Detecting Semantic Bugs in Autopilot Software by Classifying Anomalous Variables*. PhD thesis, Department of Computer Science, Tufts University, 2019.

[22] ArduPilot. *The Open Source Autopilot*. Retrieved 2017 from http://ardupilot.org/

[23] C. Lattner and V. Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. IEEE Computer Society, 2004.

[24] W. Fan, S.J. Stolfo, and J. Zhang. "The application of AdaBoost for Distributed, Scalable and On-line Learning." In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '99)* pp. 362–366, 1999.

[25] A. Broggi, E. Cardarelli, S. Cattani, P. Medici, and M. Sabbatelli. "Vehicle detection for autonomous parking using a soft-cascade AdaBoost classifier." In proceedings of the IEEE *Intelligent Vehicles Symposium*, pp. 912-917, 2014.

[26] J. Berndt. "JSBSim: An open source flight dynamics model in C++." In proceedings *AIAA Modeling and Simulation Technologies Conference and Exhibit*, p. 4923, 2004.

[27] Rife, Jason, and R. Eric Phelts. "Formulation of a time-varying maximum allowable error for ground-based augmentation systems." *IEEE Transactions on Aerospace and Electronic Systems* 44(2):548-560, 2008.

[28] Rife, Jason, and Pratap Misra. "Impact of time-correlation of monitor statistic on continuity of safety-critical operations." *Navigation* 59(4): 303-315, 2012.

[29] Rife, Jason H. "Overbounding risk for quadratic monitors with arbitrary noise distributions." *IEEE Transactions on Aerospace and Electronic Systems*, early access January 2018. Doi: 10.1109/TAES.2018.2798258.

[30] Rife, Jason H. " Comparing geometric approximations of heavy-tail effects for chi-square integrity monitors." *Navigation* in press 2018.

[31] DeCleene, Bruce. "Defining pseudorange integrity-overbounding." In *Proceedings of the 13th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GPS 2000)*, pp. 1916-1924. 2000.

[32] Rife, Jason, Sam Pullen, Per Enge, and Boris Pervan. "Paired overbounding for nonideal LAAS and WAAS error distributions." *IEEE Transactions on Aerospace and Electronic Systems* 42(4): 1386-1395, 2006.

[33] Shively, Curtis. "Derivation of acceptable error limits for satellite signal faults in LAAS." *Proceedings of the 12th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GPS 1999)*, pp. 761-770, 1999.

[34] Zaugg, T. "A new evaluation of maximum allowable errors and missed detection probabilities for LAAS ranging source monitors." *Proceedings of the Institute of Navigation Annual Meeting 2002*, pp. 187 – 194, 2002.

[35] Rife, Jason, and R. Eric Phelts. "Formulation of a time-varying maximum allowable error for ground-based augmentation systems." *IEEE Transactions on Aerospace and Electronic Systems* 44(2): 548-560, 2008.

[36] Braff, Ronald, and Curtis A. Shively. "Derivation of ranging source integrity requirements for the local area augmentation system (LAAS)." *Navigation* 47(4): 279-288, 2000.