Carving Out Control Code: Automated Identification of Control Software in Autopilot Systems

BALAJI BALASUBRAMANIAM, Department of Integrated Studies, Kansas State University, Salina, KS, USA
IFTEKHAR AHMED, Department of Informatics, University of California, Irvine, CA, USA
HAMID BAGHERI, School of Computing, University of Nebraska–Lincoln, Lincoln, NE, USA
JUSTIN BRADLEY, Department of Computer Science, North Carolina State University, Raleigh, NC, USA

Cyber-physical systems interact with the world through software controlling physical effectors. Carefully designed controllers, implemented as safety-critical control software, also interact with other parts of the software suite, and may be difficult to separate, verify, or maintain. Moreover, some software changes, not intended to impact control system performance, *do* change controller response through a variety of means including interaction with external libraries or unmodeled changes only existing in the cyber system (e.g., exception handling). As a result, identifying safety-critical control software, its boundaries with other embedded software in the system, and the way in which control software evolves could help developers isolate, test, and verify control implementation, and improve control software development. In this work we present an automated technique, based on a novel application of machine learning, to detect commits related to control software, its changes, and how the control software evolves. We leverage messages from developers (e.g., commit comments), and code changes themselves to understand how control software is refined, extended, and adapted over time. We examine three distinct, popular, real-world, safety-critical autopilots—ArduPilot, Paparazzi UAV, and LibrePilot to test our method demonstrating an effective detection rate of 0.95 for control-related code changes.

Fig. 1. QR code of the autopilot research website.

CCS Concepts: • **Computing methodologies** → **Machine learning**; Cross-validation; • **Computer systems organization** → **Robotic autonomy**; **Robotic control**; **Embedded software**; • **Software and its engineering** → *Software configuration management and version control systems*; **Software evolution**;

Additional Key Words and Phrases: Autopilot Software, Control Software, Small Uncrewed Aerial Vehicle, and Software code changes

This work was partially supported by the following awards: National Science Foundation CNS-2047971, CCF-1755890, CCF-1618132, CCF-2139845, OAC-2221648, and United States Department of Agriculture (USDA) #2023-67021-38977. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.
Authors' Contact Information: Balaji Balasubramaniam (corresponding author), Department of Integrated Studies, Kansas State University, Salina, KS, USA; e-mail: balu@k-state.edu; Iftekhar Ahmed, Department of Informatics, University of California, Irvine, CA, USA; e-mail: iftekha@uci.edu; Hamid Bagheri, School of Computing, University of Nebraska–Lincoln, Lincoln, NE, USA; e-mail: bagheri@unl.edu; Justin Bradley, Department of Computer Science, North Carolina State University, Raleigh, NC, USA; e-mail: jmbradley@ncsu.edu.

ACM 2378-9638/2024/11-ART39
https://doi.org/10.1145/3678259

ACM Reference format:
Balaji Balasubramaniam, Iftekhar Ahmed, Hamid Bagheri, and Justin Bradley. 2024. Carving Out Control Code: Automated Identification of Control Software in Autopilot Systems. *ACM Trans. Cyber-Phys. Syst.* 8, 4, Article 39 (November 2024), 20 pages.
https://doi.org/10.1145/3678259

1 Introduction

Cyber-physical systems interact with the world through complex software and physical hardware. Controllers, implemented as software, provide the primary means of enabling this interaction. Models provide guarantees about performance of the controllers, while an associated faithful implementation hopes to assure those properties. The implemented control software may only be a small portion of a potentially large software suite consisting of many components, and yet, the boundaries of the control software, and its interactions with the rest of the software suite are not always clear or well understood and may abrogate the carefully crafted guarantees. Indeed, in previous work we demonstrated that safety-critical control software may change frequently and significantly, and that these changes may impact controller performance even when the changes cannot be represented in the controller design [5].

This is especially true in small Uncrewed Robotic Systems, most commonly deployed using open source control and autonomy software.[1] ArduPilot, Paparazzi UAV, LibrePilot, PX4 and others are amongst those which were developed from the ground up in software, as opposed to a more traditional model→implementation design flow. This means models may not exist for these controllers and performance guarantees are missing, exacerbating the problem. It also means developers may not understand the significance of changes they make to the control software since the boundaries around the "control law" may be unclear. In this work we have developed an algorithm to automatically isolate changes to a larger autonomy software suite that may impact control performance, or the controller itself. The algorithm is tested on three large, popular, open-source safety-critical autopilot suites: ArduPilot, LibrePilot, and Paparazzi UAV. This capability can help developers understand the boundaries of the control software and identify what needs tested, verified, and validated in the software and improve its associated development. Ultimately, as shown in Figure 2, we imagine this tool as a part of a larger software suite that automatically identifies changes to the control software, categorizes them, and then obtains a total controller robustness score of the newly changed control software.

1.1 Research Questions

Designs, software, and hardware components for successful **Cyber-Physical Systems (CPS)** evolve over their lifetime to fix bugs, repair damage, or provide new features. In our previous work [5], we showed how quickly CPS and control software can evolve, and that many changes may not be captured in design models (e.g., exception handling), and hence not amenable to model-based design with automatic code generation [5]. But the core of the process was manual identification and categorization of the control software and associated changes. Here we advance that work

[1]The ArduPilot website reports that over one million vehicles use this code base, including companies like 3DR, Precision-Hawk, AgEagle, Insitu Boeing, Kespry, branches of the US military, and NASA among others.

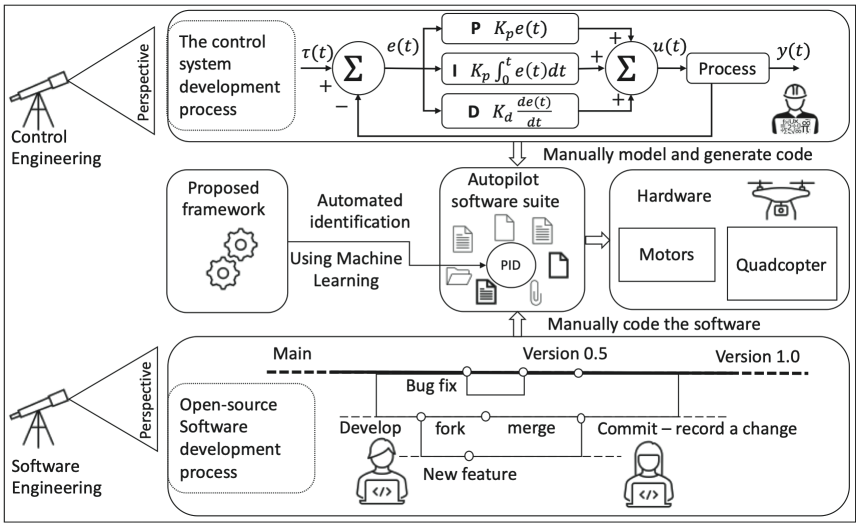

Fig. 2. Larger vision to bridge control engineering and software engineering.

by developing an automated framework to (1) identify control related commits,[2] and (2) identify maintenance changes/commits that do not impact the control software. For this purpose we define two types of commits/changes:

—Non-control commits: changes that have no impact on the controller. These changes are either for another part of the system software, or are software maintenance changes, such as documentation, refactoring, or abstraction.
—Control commits: changes that impact control software and performance.

With these definitions in mind, we address the following research questions:

RQ1: How well can we automatically identify the control related commits within projects using machine learning approaches?
RQ2: Which factors are useful in determining the control commits?
RQ3: Is cross-project training beneficial in predicting control commits?

We present an automated machine learning-based framework to identify changes and commits to the safety-critical control software with 0.95 accuracy in ArduPilot, Paparazzi UAV, and LibrePilot. This framework, in addition to answering the research questions above, addresses the key foundational opportunity to identify, extract, examine, and measure the changes made to the control software, paving the way for meeting the challenges mentioned. This framework gives us the ability to rapidly study control software evolution, helping us identify principles and rules for the creation of future tools that incorporate this evolution for controller development and maintenance. Because we can identify the boundaries of control software and associated changes, we can develop software engineering and control design strategies to produce more robust controllers to software changes.

Figures 3 and 5 depict a high-level overview of our automation framework to address the first and third research questions, which consists of analysis steps and employing machine learning

[2]In version control systems, such as Git, **Subversion (SVN)**, a commit is an operation which sends the latest software code changes of the original source code to the software repository, each such change is logged by commit **Identity (ID)**s that are identified by unique hash values. This commit ID is created whenever a new commit is recorded and provides traceability benefits. Hence, we use this commit as our fundamental unit for our categorization.

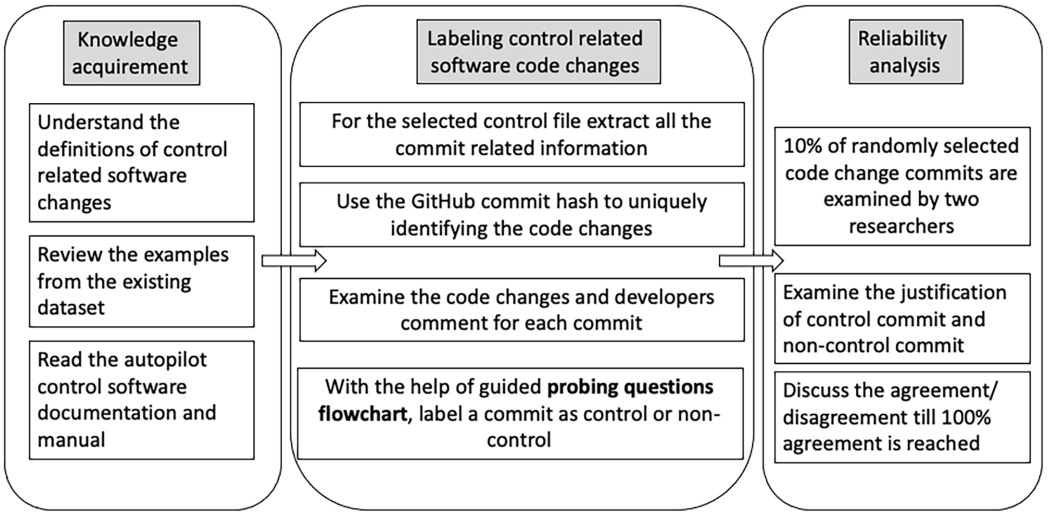

Fig. 3. Manual labeling process.

algorithms for classification. The process to answer RQ2 starts by systematically analyzing the code changes that are responsible to identify control related commits and then organize them in a mindmap.

The remainder of the article is organized as follows. In Section 5 we mention existing work in mining code changes and empirical studies of coding practices and automation. Section 2 describes the tool that we developed and the methodology of our empirical study. In Sections 3 we talk about the results of our study and their implications, in Section 4 we list possible threats to validity of our research, and in Section 6 we draw up conclusions and discuss possible directions for future work.

2 Methodology

Our goal is to automate the identification of control code changes. We collect a representative corpus of code changes from the autopilot controller software projects mentioned. These code changes are examined by five different machine learning classifiers. We use the following process during our study: (a) we collect a sample of control files from the autopilot controller software projects; (b) we extract all the code changes from these control files; (c) we label each and every code change as either control or non-control related change; (d) we conduct supervised training with five different machine learning classifiers and compare the results; and (e) we repeat the above steps for cross-project control prediction. We describe each of these steps in further detail in the following subsections.

2.1 Project Selection

The prime aim of the project selection was to: (1) analyze autopilot system that is representative of code developed in the real world, (2) include significant safety-critical control code changes with many available versions reflecting their software development process, and (3) support reproducibility of the results obtained from our methodology. Therefore, we only select active, open-source autopilot software projects from GitHub. In addition, we opted to select projects that use programming languages C/C++ for two reasons: first, embedded systems widely use C or C++, and second, "compiled" languages are extremely efficient as they are fast and stable making them often more suitable for low-level control.

We began by selecting two autopilot software projects—ArduPilot [4] and Paparazzi UAV [41]. PX4 [48], another very popular open-source autopilot, was, in part, derived from ArduPilot, having collaborated to produce the full flight control stack.[3] As a result, we have not selected it in this work, believing it might not add meaningfully to our results since PX4 and ArduPilot may share common design, file, and style elements. Additionally, ArduPilot is more mature, having been in development for longer (circa 2009); PX4 did not exist till 2012. We analyzed ArduPilot and Paparazzi UAV projects that have more than 10 years of development history and can operate a variety of vehicles including airplanes, multi-rotors, helicopters, and boats. These repositories have 448 and 99 contributors, respectively. Also, they contain ~38,000 and ~15,400 code changes, respectively. The latest version contains approximately 441k **lines of code (LOC)** and 1.3M LOC, respectively.

Our previous research [5] demonstrated that small code changes can dramatically affect a controller's behavior using a mutation tool. It uses abstract syntax tree parsing to change the software code locations, such as the control variables. We quantified the control performance algorithm via eight traditional control step response metrics: rise time, settling time, settling min, settling max, overshoot, undershoot, peak, and peak time. And found that small code changes can dramatically affect a controller's behavior. In the current work, our novel framework addresses the key foundational opportunity to identify, extract, examine, and measure the changes made to the control software, paving the way for meeting future challenges. To facilitate the rapid study of control software, we built a machine learning tool. Automating this classification process will help facilitate a more rapid study of control software.

To test the cross-project evaluation of our proposed automatic classification, we selected a third project—also widely used autopilot software—LibrePilot [32]. It provides a sophisticated control system that supports various flight modes and can operate on a variety of vehicles including multi-rotors, fixed wing, and cars. It has over 11 years of well maintained history, and its code base is accessible through a git repository that stores the code changes committed by the developers since June, 2010. As of October 2021, the repository includes 66 contributors that have committed almost 15,000 changes. The latest version of LibrePilot contains approximately 452k LOC in C/C++. The details of the three autopilot software projects are presented in Table 1.

Although "control" can be more broadly defined when considering a full autonomy stack that includes successive loop closure, guidance, navigation, and planning, we focus on, and limit our analysis to the control files that provide coverage of functionality associated with position and attitude control. We selected a total of 15 control files central to position and attitude control from the three different autopilot software projects. We then analyzed a total of 1,435 commits, 471 commits from LibrePilot and 964 commits from ArduPilot and Paparazzi UAV, where each commit included changes to at least one of the target files.

2.2 Manual Labeling

Identifying the code changes that impact the control is not a trivial task as all code changes either directly or indirectly impact the control of the vehicle. Adding to the complexity of identifying control related code changes, a commit can change more than one line of code in more than one file. In software engineering, we capture the change occurring in a file using a measure called code churn [21]. To further highlight the implication, we use the "Rewrite Rate" metric to capture how often the original controller software has essentially rewritten. It is computed as $\frac{Z}{Y}$, where Z is the code churn and Y is LOC changed in the earliest commit. From our previous work [5], we observed that half of the analyzed files represent "Rewrite Rates" of around or above 20, indicating those

[3]The history of PX4 website URL—https://auterion.com/company/the-history-of-pixhawk/

Table 1. Details of Three Autopilot Software Projects

Repository name	Total commits	Development duration	LOC	Code profile [43]
ArduPilot	63,066	13 years	441k	C++: 63% Python: 12% C: 10% Objective-C: 9% Lua: 3% HTML: 1% MATLAB: 1%
Paparazzi UAV	16,339	10 years	1.3m	C: 76% OCaml: 7% Python: 7% Perl: 3% Makefile: 2% C++: 2% Scilab: 1% MATLAB: 1% Processing: 1%
LibrePilot	15,662	11 years	452k	C: 59% C++: 34% Fortran: 3% Assembly: 1% HTML: 1% Python: 1% Makefile: 1%

control files have almost nothing in common with the original versions. To give perspective, even the file with the lowest growth rate, AC_PosControl.cpp, has been rewritten almost three times.

To manually label a code change as a *control*[4] category we followed the process in Figure 3. The labeling process is performed in three phases: (1) knowledge acquirement, (2) labeling control related code change, and (3) reliability analysis. In the knowledge acquirement phase, we gathered the relevant knowledge required to identify control related software change. The first step in this phase is to understand the definitions of control category code change. To further enhance the knowledge we also looked at the definitions and examples from our previous work [5] for better understanding. As an additional resource, online documentation about the control code was used.

The second phase involves examining the code change commits. We used the following materials for our guidance—autopilot software manual and documentation, developer comments, and aerospace drone/control keywords. With this information, each commit will be analyzed as shown in Figure 4, based on the probing questions [3] labeling flowchart. We identified all changes and tried to determine whether or not a change impacts the control code in some way. If it impacts the control code then we label it as "control." Otherwise, we label as "non-control." This may not always be a straightforward process as some changes may indirectly impact control code. To mitigate this we manually analyzed the code changes to see the impact on control code. Finally, if the code change impacts the control of the vehicle we label that commit as *control* and anything else was put into the *non-control* [5] category. This process is repeated for all the commits. The total number of *control* and *non-control* commit instances in the three autopilot software projects are presented in Table 2.

Finally, to understand the agreement between the different researchers we use 10% of the commits to validate our findings between the authors. From the pool of 471 commits, we extracted and labeled a random sampling of 47 commits (approximately 10% of all commits). This random sample represents the code changes in three different autopilot software projects. We had 93% agreement and 100% agreement in the control-labeled commits and non-control-labeled commits, respectively

$$\kappa = \frac{Pr(O) - Pr(C)}{1 - Pr(C)}, \tag{1}$$

[4]We use a different font, sans-serif typeface italic shape, to distinguish the use of the word control as a label name.
[5]Similar to control label, we use a different font, sans-serif typeface italic shape, to distinguish the use of the word non-control as a label name.

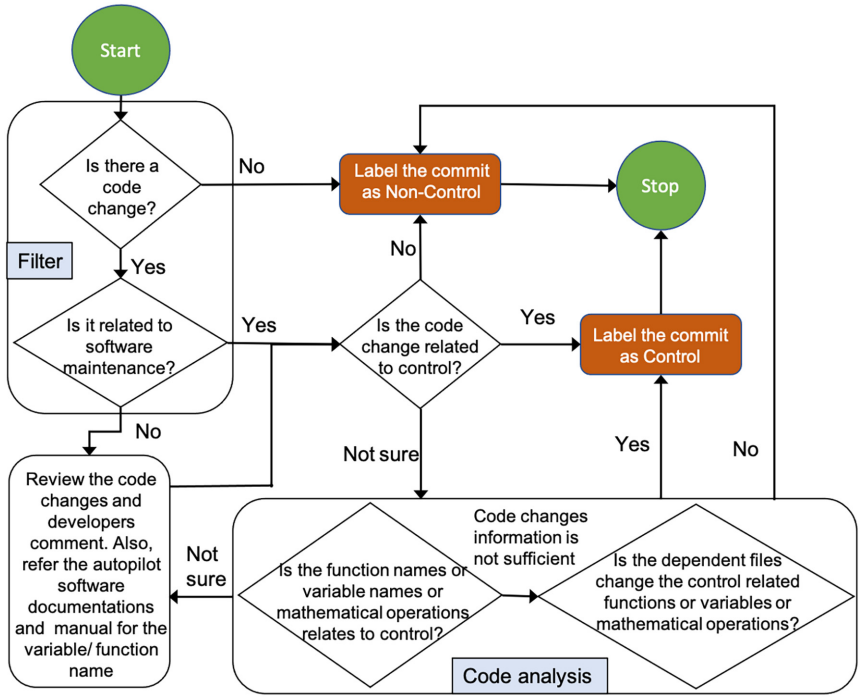

Fig. 4. Probing questions approach.

Table 2. Number of Control and Non-Control Commit Instances in the Three Different Autopilot Software Projects as of October 22, 2022

	ArduPilot	**Paparazzi UAV**	**LibrePilot**	**Total**
Non-Control	189	225	180	594
Control	396	154	291	841
Total	585	379	471	1435

calculates the degree of agreement between authors during the manual labeling process. We use the inter-rater reliability metric using Cohen's, κ. In this equation, $Pr(O)$ represents the actual observed agreement, and $Pr(C)$ represents the chance agreement. As Cohen's kappa κ takes into account the rater agreement due to chance. We adopted the chance agreement calculation from Mary L. McHugh [36]. Our manual analysis of Cohen's kappa is 0.84, which can be interpreted as almost perfect agreement. After the first and second iteration, we had 98% agreement and 100% agreement in the control labeling, respectively.

The difficulty in the manual labeling process is to hastily base the classification on a single metric or one aspect of the commit rather than understanding the entirety of the code change and its impact on the control of the vehicle. For example, as shown in Listing 1, we highlight a deceptively simple code change, the renaming of `Airspeed` to `TrueAirspeed`[6] in the `pathplanner.c` file. However, in reality, this single commit changed a total of 11 files with 55 code additions and 38 code deletions.

[6]"True airspeed" is the idealized airspeed of an aircraft used for planning purposes.

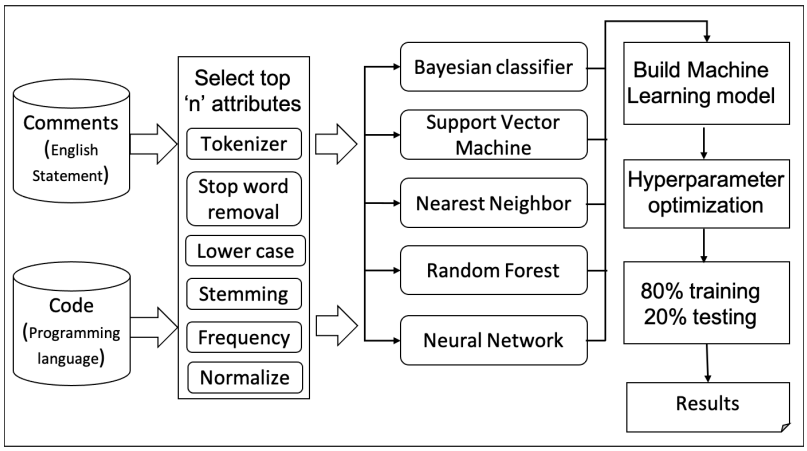

Fig. 5. Automated machine learning-based framework to identify the safety-critical control software changes.

Based on the developer's comment and code change, one may be tempted to label this code change as *non-control* as it is a simple rename. However, upon further inspection, based on the probing questions approach, in the same commit, we find that there is another file called `baroairspeed.xml` in which the data for this variable is actually changed. In this case, the `baroairspeed.xml` file already had both the `Airspeed` parameter and `TrueAirspeed` parameter. After the code change, the developers renamed `Airspeed` to `CalibratedAirspeed`[7] in `baroairspeed.xml` file. But in the `pathplanner.c` file calculation, however, they renamed `Airspeed` to `TrueAirspeed`. Hence, the velocity calculation end result is impacted because the data is potentially significantly different.

```
BaroAirspeedData baroAirspeed;
BaroAirspeedGet (&baroAirspeed);
if (baroAirspeed.BaroConnected == BAROAIRSPEED_BAROCONECTED_TRUE) {
    -  velocity = baroAirspeed.Airspeed;
    +  velocity = baroAirspeed.TrueAirspeed;
}
```

Listing 1. Code changes in papthplanner.c file

2.3 Machine Learning

Our objective is to perform extensive analysis on a large scale, requiring an automated method. Through manual analysis, we found that basic keyword or rule-based techniques struggled to accurately identify control-related code changes. Traditional software engineering techniques take the rules-based approach to solving problems. Machine learning techniques, on the other hand, learn from data and solve problems. Recognizing the limitations of such approaches, we opted to implement a machine learning-based method, which offers more sophisticated and accurate identification capabilities.

To automate the identification of control code changes, as shown in Figure 5, we used five machine learning techniques: Bayesian classifier [53], **support vector machine (SVM)** [46], nearest neighbor [2], **random forest (RF)** [10], and neural network [39]. We used a wide range of learning techniques to reduce the risk of dependence on a particular algorithm or implementation. We observed that the selection of top 'n' feature attributes included common control keywords (e.g., control, derivative, error, feedforward, filter, frame, frequency, gain, integral, kalman, proportional)

[7]"Calibrated airspeed" is indicated, or measured airspeed corrected for instrument and positional errors.

and also keywords specific to the target autopilot controller (e.g., acceleration, altitude, distance, pitch, roll, yaw, waypoint, speed, velocity). We used the Auto-WEKA [28] with 10-fold cross-validation for evaluation.

This binary classification approach for discerning control-related code changes offers both novelty and practical utility. Traditionally, identifying such changes demanded intricate, time-consuming methods like manual scrutiny or complex rule-based systems. However, with a straightforward binary classification model, this process becomes streamlined and automated, yielding significant time and resource savings. This model expedites accurately detecting control-related changes, enhancing software maintenance, debugging, and enhancement endeavors. Moreover, its simplicity facilitates seamless integration into existing development workflows, empowering software engineers and developers to manage and monitor control code alterations effectively. Notably, the model's feature selection enables users to understand the classification rationale, enhancing its real-world applicability.

2.3.1 BayesNet. We use the Bayesian Network classifier [53] that applies Bayes rule, both qualitatively and quantitatively, to classify our code change commits either into *control* or *non-control*. Intuitively, the qualitative component of a Bayesian network explicates features in the code change commits and captures the direct influences between them. The quantitative component in the form of conditional probabilities quantify the dependencies between features from code change commits. We also used the Grid search technique to perform hyperparameter optimization for different classifier configurations. Hyperparameter optimization directly controls the behaviors of training algorithms and significantly affects the performance of machine learning models [56]. We varied two parameters: (1) alpha is used for estimating the probability tables and can be interpreted as the initial count on each value, and (2) the maximum number of parents in the Bayes net.

2.3.2 SVM. Based upon an assumption that code changes would be linearly separated across factors, we selected SVM [46]. Our SVM uses the standard Radial Bases Function Kernel [29] and for the other parameters we performed a grid search to choose the best classification. This configuration result had a gamma ($gamma) ranging from 0.1 to 0.0001, a complexity C ranging from 1 through 100 and one maximum iterator.

2.3.3 RF. Since RFs do not overfit because of the Law of Large Numbers, they are an effective tool in prediction [10]. Injecting the right kind of randomness makes them accurate classifiers. We configured the maximum depth of the tree from 0 to 10 and also the number of features that sets the number of randomly chosen attributes. We configured a batch size of 100 that determines the size of the instances used for training a classifier.

2.3.4 Nearest Neighbor. We used **instance-based k (IBk) k-Nearest Neighbor** classifier, which uses similarity of the closely related instances to make predictions [2]. We selected this classifier technique as it makes no assumptions. It uses knowledge of the nearest data points to predict the output and hence used as a first option when there is a little or no prior knowledge. We configured the number of neighbors from 0 to 100 and the window size to get the maximum number of instances allowed in the training.

2.3.5 ***Multi-Layer Perceptron (MLP).*** We used an MLP to see if it could leverage hidden relationships not explored in the other algorithms [39]. We configured our Perceptron with a 0.3 learning rate, a 0.2 momentum, and 500 epochs. The Perceptron would terminate its validation testing after not being able to reduce its error 20 times in a row.

We also used Auto-WEKA [28] for identifying the best classifier, which automatically searches through the joint space of WEKA's learning algorithms and their respective hyperparameter

settings to maximize performance, using sequential model-based optimization [11] (a Bayesian optimization method). In Table 3, we provide hyperparameter details. Though there is one Python based implementation called Auto-sklearn [16], we chose Auto-WEKA because it comprises a larger space of models and hyperparameters [28] compared to Auto-sklearn. As a result, RandomForest and SVM were determined to be the best techniques. These results were generated by running Auto-WEKA with random seed 123 for 5 hours.

2.3.6 Training and Testing. We applied five different machine learning techniques and Auto-WEKA using 10-fold cross validation and report the results in Table 4. A total of 1,435 commits, 471 commits from LibrePilot and 964 commits from ArduPilot and Paparazzi UAV was used in training and testing. Since 1,435 commits are extracted from three different autopilot software projects, the Table 4 results also shows the cross-project prediction. This is important for some projects, specially projects that do not have historical data to perform any significant training. Hence, we investigated whether it is feasible to perform cross-project training following the method used by Rahman et al. [49].

2.4 Measures

We report the standard precision, recall, **false positive rate (FPR)**, and Area Under the receiver operating characteristic Curve (AUC) to asses the performance of the prediction models, because it is independent of prior probabilities [7]. Also, AUC is a better measure of classifier performance than accuracy because it is not biased by the size of test data. Moreover, AUC provides a "broader" view of the performance of the classifier since both sensitivity and specificity for all threshold levels are incorporated in calculating AUC. Other work related to prediction have used AUC for comparison purposes [14, 18, 19, 58]. We list the formula used for calculating precision, recall and FPR below. Based on the machine learning techniques, we get varying rates of false positives/negatives (f_p/f_n), and correct predictions in terms of true positives/negatives (t_p/t_n). The AUC curve is created by plotting the recall against the FPR at various threshold settings.

Precision (P): A measure of whether the *control* predictions were relevant

$$precision = \frac{t_p}{t_p + f_p}. \tag{2}$$

Recall (R): A measure of the percentage of *control* instances that were correctly retrieved by prediction models

$$recall = \frac{t_p}{t_p + f_n}. \tag{3}$$

FPR: A measure of the ratio of the number of *control* commits wrongly categorized and the total number of actual *control* commits

$$FPR = \frac{f_p}{f_p + t_n}. \tag{4}$$

3 Evaluation and Results

In this section we discuss the results of our study by placing them in the context of three research questions, which investigate the ability to predict the control commits (RQ1), factors that are useful in determining the control commits (RQ2), and whether we can perform cross-project control commit prediction (RQ3).

Table 3. Grid Search Technique Used to Perform Hyperparameter Optimization for Different Classifiers Configuration

Classifier name	Parameter interval	Parameter description
Bayesian Network (BayesNet)	alpha = [**0.5**,1,1.5,2]	alpha is used for estimating the probability tables and can be interpreted as the initial count on each value.
	MaxNrOfParents = [**1**,2,3,4,5]	MaxNrOfParents sets the maximum number of parents a node in the Bayes net can have.
SVM (SMO_RBFkernel)	Complexity=[**1**,10,100]	Complexity parameter.
	Gamma=[0.1,**0.01**,0.001,0.0001]	The gamma value.
RF Decision tree (J48)	Confidence factor=[0.15,**0.25**,0.35,0.45]	The confidence factor used for pruning (smaller values incur more pruning).
	minNumObj=[1,**2**,3,4,5]	Minimum number of instances per leaf.
Nearest Neighbor (IBk)	KNN=[**1**,2,3,4,5]	Number of neighbors to use.
	maxInstLeaf=[20,30,**40**,50,60]	The maximum number of instances in a leaf.
MLP (NN)	H=[1,10,100]	Number of hidden layers. Default:251.
	learning rate = [0.1,0.2,**0.3**,0.4,0.5]	The learning rate for weight updates.
	Exponent = [**1**,2,3]	Exponent controls the degree of the polynomial. Set to 1 for the linear kernel. Set to 2 for the quadratic kernel and 3 for the cubic kernel.
Auto-WEKA (5 hours)	automatically performs hyperparameter optimization	

Parameters in bold are the default. Parameter description adapted from WEKA documentation.

3.1 Results for RQ1: How Well Can We Automatically Identify the Control Related Commits within Projects Using Machine Learning Approaches?

To answer our first research question as to whether we can automatically identify control code, we used the 964 commits from ArduPilot and Paparazzi UAV and the newly added 471 commits from LibrePilot. We applied five different machine learning techniques using 10-fold cross validation: Bayes network (BayesNet), SVM, RF, NN, and MLP. We also used Auto-WEKA, which automatically searches through the joint space of WEKA's learning algorithms and their respective hyperparameter settings to maximize performance and identify the best classifier. The algorithm with the best performance according to Table 4 is RF, which has the highest AUC of 0.95. Table 4 shows the results in terms of precision, recall, and AUC.

As shown in Table 4, for *control* commits, the recall was 93% in comment based classification. 779 out of 841 commits were correctly predicted as *control* which were the commits of interest in

Table 4. Results of Automatic Identification of Control Code Changes in ArduPilot, LibrePilot and Paprazzi UAV

	Class	Precision	Recall	AUC
Comment (English sentence)	Non-control	0.86	0.65	0.87
	Control	0.79	0.93	0.87
	Weighted Avg.	0.82	0.81	0.87
	Class	**Precision**	**Recall**	**AUC**
Code (software code)	Non-control	0.82	0.98	0.95
	Control	0.99	0.85	0.95
	Weighted Avg.	0.92	0.90	0.95

this study. Similarly, the recall was 85% in code based classification. 711 out of 841 commits were correctly predicted as *control*; this is comparatively less than comment based classification. But the precision was 99% indicating that the prediction of *non-control* as *control*, statistically called as false positives, were reduced. AUC is the measure of a classifier's ability to distinguish between classes. The higher the AUC value of a classifier, the better the performance of the classifier model at distinguishing between *control* commits and *non-control* commits. Our results have a higher AUC (95%), indicating our best model has a better performance at classifying the *control* commits.

3.2 Results for RQ2: Which Factors Are Useful in Determining the Control Commits?

To examine the useful factors in determining the *control* commits, we provide our observations and highlight the important facts in a mindmap. In a broader sense, we are aware of two changes: (1) software code change and (2) control code change. The software changes include renaming, refactoring, documentation, moving code to library or other files, print statements for debug, and moving parameter values to settings files for flexible configuration. The control changes include modifications to K_p (proportional component), K_i (integral component), and K_d (derivative component) value changes, resizing oversampling window for sensor data, normalize vector, better filtering, calculation change and coefficients change. However, our article aims to shed light on the control changes that are often overlooked.

Overlooked control code changes can have an indirect impact on control, such as adding flightplan safety checks to pathplanner, a new motor definition for OctoX body frame, an added flight plan cyclic redundency check consistency checks, resizing stack sizes that execute tasks in real time operating system, using callbacks on the waypoints, and updating desired position. To help characterize these types of changes we provide these observations in a mindmap, as shown in Figure 6. This can be used to enhance the understanding of autopilot control code changes. Recent research has found that mindmaps can be effective as a learning tool [42, 55]. In the following section, we discuss how the mindmap has visually organized the autopilot code change information with examples.[8]

Software changes: This is the easiest to understand and also has no impact on the overall control performance. For example, in commit ID 5484, line 217, the developers have added a standard error print statement to check the work-in-progress code while building simulation firmware for OSX. As part of this commit, they also used PiOS core hardware abstraction layer to compare raw times

[8]We refer to the examples with commit ID; this directly references real-world developer code changes. You can review these changes and download them by visiting our autopilot research website by following this URL—https://controlsoftware.autopilotresearch.com/CI/controlleridentification.html

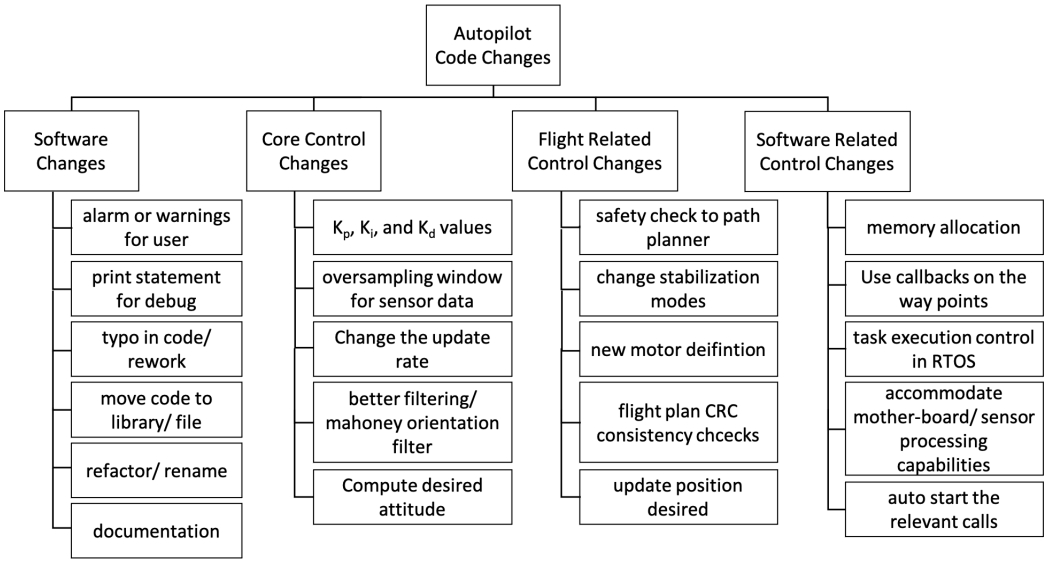

Fig. 6. MindMap of autopilot control code changes.

and convert them to a microsecond value. However, they did not use PiOS wait which provides a micro-second granular delay using the CPU cycle counter. Hence, this code change has no logical control impact.

Core control changes: These changes are typical control software changes. For example, in commit ID 5390, line 137, the task delay is changed, and in line 54 the waypoint is updated. This change relates to time and space operation of the control and impacts it directly. A control software developer should be able to identify this. The delay of the *pathplanner* task changes from 100 ticks to 20 ticks, effectively the frequency at which the function is called. The waypoints are updated more frequently from the ground control station impacting the control response and performance.

Flight related control changes: The challenging part of this work is to identify the indirect changes, such as affecting the simulation or changing the mode of operation. For example, in commit 6,335, line 576, the change allows easy switching between indoor and outdoor mode. As a result only the magnetometer configuration is used for flying without setting the home location impacting the overall sense of vehicle direction and mission completion between various mode of operation.

Software related control changes: These are the most difficult to identify and often overlooked or assumed to be solved by the control community. These code changes impact the control indirectly, such as task order execution, computer architecture memory allocation, but are more subtle and may require deeper understanding of the interaction of computing and control. For example, in commit ID 5520 as provided in the Listing 2, in line 252, this commit introduces a 1 ms timeout to avoid a race condition. During attitude estimation the *accel* queue would not immediately have the data available. The change allows the tasks to wait until the gyroscope data and acceleration data are available for attitude estimation.

```
// Wait until the AttitudeRaw object is updated, if a timeout then go to
    ↪ failsafe
- xQueueReceive(accelQueue, &ev, 0) != pdTRUE )
+ xQueueReceive(accelQueue, &ev, 1/ portTICK_RATE_MS  )!=pdTRUE)
```

Listing 2. Commit ID 5520 - Code changes in attitude.c file

Table 5. Results of ArduPilot and Paprazzi UAV as Training and LibrePilot as Testing for Cross-Project Prediction Analysis

	Class	Precision	Recall	F-Measure	AUC
Comment (English sentence)	Non-Control	0.54	0.38	0.45	0.67
	Control	0.68	0.80	0.73	0.67
	Weighted Avg.	0.63	0.64	0.62	0.67
	Class	**Precision**	**Recall**	**F-Measure**	**AUC**
Code (software code)	Non-Control	0.67	0.50	0.57	0.65
	Control	0.73	0.84	0.76	0.67
	Weighted Avg.	0.71	0.71	0.70	0.66

As an additional example, in commit ID 14182, in line 141, the developer changed the data type to use *enum* instead of uint8_t. There are two likely reasons [20, 50]: (1) The size of an enum is implementation specific; if it is user-defined, you can control the size. This can be important for small buses, small memory systems, or to minimize space when serializing to a disk or a socket. (2) If the information is used in another scope, passed across code boundaries, stored or serialized in some form that requires a specific type, or defined as a bitmask then an enum can improve performance. Especially in embedded code where memory is tight, this technique can improve the data cache hit rate if many accesses are necessary. In a safety-critical autopilot system, where timing is critical, this can heavily impact control performance. This single commit has 14 changed files with 48 additions and 43 deletions.

Our mindmap with the above discussed observation examples helps to make new connections between useful factors to ascertain the control commits and control code changes that are often overlooked. To enhance the understanding of autopilot control code changes, we organized this information into four categories based on the factors in determining the control commits.

3.3 Results for RQ3: Is Cross-Project Training Beneficial in Predicting Control Commits?

In RQ1, we tested our machine learning algorithms using a 10-fold cross-validation with all our data (1,435 commits). However, considering other open source autopilot software projects that may not have historical data to perform any significant training, we perform cross-project prediction analysis by training the algorithms on the dataset from our previous work [5] (this dataset contains ArduPilot and Paparazzi UAV) and then testing the resulting classifier on the newly added LibrePilot dataset. Cross-project prediction analysis has been developed and investigated in other areas of software engineering, such as defect prediction [33, 52, 60]. However, investigating the applicability of cross-project control commits is novel. We followed the method used by Rahman et al. [49] to perform cross-project control commit prediction analysis. To test the portability, Rahman et al. evaluated machine learning models trained on one project on all releases of other projects, ignoring time-ordering. We adopt this method by considering ArduPilot and Paparazzi UAV as one project and the added LibrePilot as a different project to perform cross-project control commit prediction analysis.

Table 5 shows the result of our best performing machine learning algorithms using Auto-WEKA for different sets of evaluation metrics (precision, recall, F-Measure and AUC). We added the F-Measure metric to this test as it provides intuition on how precise the classifier is. The overall prediction performance difference is considerable compared to RQ1 with AUC of 0.67 and an

F-Measure of 0.76. Although we can train across projects, the comment based prediction and code based prediction had a performance decline compared to RQ1.

Our findings, based on precision, recall, F-Measure and AUC, indicate that the comment based prediction seems to be less stable compared to code based prediction. One of the reasons may be that all the developers involved in ArduPilot, Paparazzi UAV, and LibrePilot are different in the comment based prediction. As a result, their writing style differed from one developer to another. Moreover, checking for incomplete sentences or verification of meaning in the sentences are not strictly enforced. It is also important to note that most of the comments are short and not as detailed as they might be in a manual or other documentation. On the other hand, compared to comment based prediction, the code changes present in the code based prediction had valuable details because: (1) codes must follow the strict rules of programming, (2) codes must be checked for errors and testing results, and (3) codes may often provide additional information about the characteristics of the surrounding program code. This suggests leveraging a cross project approach using code has fewer factors tied to influence machine learning model prediction.

4 Threats to Validity

We have taken care to ensure that our results are unbiased, and have tried to eliminate the effects of human errors. Nevertheless, in this section we discuss threats to validity for our study.

External validity: These are threats that concern the generalization of our findings. Our research findings are focused on three autopilot control software projects. This choice was opportunistic in that ArduPilot, Paparazzi UAV, and LibrePilot have been widely deployed and are open-source, so findings in these code bases can still be valid for similar systems (e.g., PX4 [45], BetaFlight [8]). Likewise, even though the cost of analyzing hundreds of commits limited the scope of files studied, those files perform different controller tasks and were designed by different groups of developers. As a result, we anticipate these findings will also apply to other files designed by other developers. We also acknowledge that the granularity of changes we studied (i.e., commits) may not expose all code changes made by developers.

To support reproducibility: (1) we have used an open source version control and source code management functionality, GitHub, (2) we used an open source knowledge analysis tool that has the collection of machine learning algorithms, WEKA, and (3) developed a project website hosting all the labeled datasets and machine learning models. Researchers can download our machine learning models[9] and use our database to get the same results and possibly apply on new auto pilot software projects. We have also included the QR code in Figure 1 in the abstract section to visit our digital resources more quickly and efficiently.

Internal validity: These are threats that concern uncontrolled factors that may have affected our results. Since we used our manual analysis approach to figure out if a code change impacts the control code or not, we may have missed some of the control code changes. However, we took precaution by reviewing the code changes by multiple researchers to label the code changes as *control* or not. Such judgment calls are subject to many biases, which we tried to reduce by defining clear criteria for filtering and classification, by having multiple authors check different parts of the results, and by iterating and revisiting the results. However, it is still possible we may have missed or mislabelled some of the control code changes as *non-control*. To overcome this, we also used the inter-rater reliability using Cohen's Kappa κ to show the manual label agreement by chance. We have explained the Kappa calculation process and agreement results in our methodology section. Our manual labeling agreement was between different researchers with 10% of the commits to

[9]You can review and download the datasets and Machine Learning models by visiting our autopilot research website by following this URL—https://controlsoftware.autopilotresearch.com/CI/controlleridentification.html

validate our finding, and the percentage agreement is 0.93 while the Kappa is 0.84—a considerable reduction; the greater the expected chance agreement, the lower the resulting value of the kappa. This can be interpreted as almost perfect agreement.

Construct validity: These are threats that concern our metrics and measures. Regarding the evaluation of the classification techniques, we used the standard performance metrics precision, recall, F-measure, and AUC. We used widely-used autopilot software projects along with a large dataset of *control* commits (841) and *non-control* commits (594), which has preserved code changes for more than 10 years. Although this cannot serve for forming universal claims, it injects realism in our analysis as the control commits have been delivered in widely used real-life projects with elaborate complex functionalities delivered to hundreds of thousands of users.

5 Related Work

In this section, we provide a discussion of the related efforts in the control software area and examine them in the light of our research.

5.1 Safety and Machine Learning

Machine learning algorithms are increasingly used in the control system of autonomous cyber-physical systems, such as self-driving cars, to make intelligent navigation decisions in real-time without any human input. Varshney et al. [54] discuss strategies for increasing safety in terms of harm, risk, and uncertainty and building upon it in the machine learning context. Also, Khan et al. [26] introduce a rigorous method of machine learning for secure and autonomous cyber physical systems. Pereira et al. [44] present the challenges during the design and development of a system with an integrated Machine Learning algorithm. They also discuss the approaches for safety engineering and certification of Machine Learning-based systems. Different from all prior work, this article focuses on using machine learning algorithms during the control software development process to provide insight into the evolution of control software and the impacts thereof.

5.2 Bug Detection and Machine Learning

Building bug-free autopilot control software continues to be a very significant challenge. Huang et al. [23] apply Machine Learning for runtime bug detection in aviation software. Kim et al. [27] presents a cross-domain robotic aerial vehicles post-accident investigation tool that localizes program-level root causes of accidents, based on robotic aerial vehicles control model and enhanced in-flight logs. Even though our technique does not directly target detecting bugs, it takes important first steps in automating the identification and isolation of the safety-critical control-related software in, by and large autopilot software projects.

5.3 Model-Based Design Strategies

Model-based design strategies ideally create a 1:1 correspondence between the model and the software [15, 47, 59]. This is done for safety-critical systems by: (1) building models in a numeric computer environment (e.g., MATLAB, Simulink, Stateflow, or other tools), (2) verifying these models, and then (3) autogenerating corresponding code. The code synthesis tool often must be certified and adhere to standards (e.g., DO-178C) [13, 51] to produce provably correct results. While this strategy links the model and autogenerated software, it may only exist in domain-specific applications [25], and may not provide guarantees for third-party supporting software libraries, drivers, or other specialized pieces of code used in the development of the system, or may be otherwise incomplete. In addition, the vast majority of autopilot control systems, including unmanned air systems, are typically not developed using a model-based design strategy, but rather, use hand coded controllers, such as ArduPilotfor which there may be no mathematical or computer

model at all. Hence, our work could potentially bridge the gap by identifying the control related code changes during the development process.

5.4 Security and Vulnerability

As unmanned robotic system usage is increasingly popular, another set of researchers aims to solve the challenges of malicious attacks and mitigate system vulnerability [1, 37, 57]. Mohamed et al. [38] authors discuss the usage of advanced cyber penetration tools for criminal network surveillance. Another vital source code analysis technique is taint analysis [12, 40]. It helps to identify the flow of sensitive data through a program. Taint analysis could be a complementary technique to our current work. One such application uses a watch list of control variables to potentially capture core control changes shown in Figure 6. However, we may miss out on flight-related control changes and software-related code changes. These overlooked code changes can indirectly impact control. Controllers, such as Proportional, Integral, and Derivative, serve as the heart of controlling the movement of vehicles. Our proposed framework can serve as a first step in identifying these control code changes.

5.5 Software Engineering

The software engineering community has developed a wide range of techniques to cope with the challenges involved in validating and verifying the safety and security of safety-critical systems, including control software (e.g., [9, 22, 30, 34, 35]), or their sound applications to assist in self-adaptation [17]. Among others, Jackson [24] discusses a new architectural pattern called certified control that is used for achieving high assurance of safety in autonomous cars. Simone et al. [31] and Peng et al. [6] discuss embedding adaptation features in the ArduPilot control structure, and their software-in-the-loop experiments show improved performance. Unfortunately, outside of highly regulated safety-critical systems, the application of the majority of these strategies is limited due to their high costs. This is particularly noticeable in the extremely active Unmanned Air System industry where open source autopilot control systems (e.g., ArduPilot [4], Paparazzi UAV [41], LibrePilot [32], PX4 [45]) are used extensively on various types of hardware with contrastingly very light, or no regulations in their design and test processes.

6 Conclusions and Future Work

Identifying safety-critical control software, its boundaries with other embedded software in the system could help developers isolate, test, and verify control implementation, and improve control software development. In this work, we have directly studied the code changes in three different dominant open-source control software suites, ArduPilot, Paparazzi UAV, and LibrePilot, used extensively in safety-critical Unmanned Aircraft System. Our novel framework addresses the key foundational opportunity to identify, extract, examine, and measure the changes made to the control software, paving the way for meeting future challenges. To facilitate the rapid study of control software evolution we built a machine learning tool that can automatically identify the control code change with an accuracy of 0.95.

In future work, we hope to develop design-time and runtime solutions that isolate impacts of verified autogenerated control code from code external to that process and vice versa. As we learn more about control code changes we envision a supplementing set of control design strategies and theory for how to build improved computer-controlled systems, an increase in developer tools suited for control code, and improved test suite generation. A deeper understanding of the types and quantity of control changes can help the control and software communities develop new models and development strategies to maintain the integrity of key properties verified in the model and/or software.

References

[1] Dennis Agnew, Alvaro Del Aguila, and Janise McNair. 2024. Enhanced network metric prediction for machine learning-based cyber security of a software-defined UAV relay network. *IEEE Access* 12 (2024), 54202–54219.

[2] David W. Aha, Dennis Kibler, and Marc K. Albert. 1991. Instance-based learning algorithms. *Machine Learning* 6, 1 (1991), 37–66.

[3] Preethu Rose Anish, Balaji Balasubramaniam, Abhishek Sainani, Jane Cleland-Huang, Maya Daneva, Roel J. Wieringa, and Smita Ghaisas. 2016. Probing for requirements knowledge to stimulate architectural thinking. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. IEEE, 843–854.

[4] ArduPilot. 2024. ArduPilot Open Source Autopilot. Retrieved from https://ardupilot.org/

[5] Balaji Balasubramaniam, Hamid Bagheri, Sebastian Elbaum, and Justin Bradley. 2020. Investigating controller evolution and divergence through mining and mutation. In *Proceedings of the ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS '20)*. IEEE, 151–161. DOI: https://doi.org/10.1109/ICCPS48487.2020.00022

[6] Simone Baldi, Danping Sun, Xin Xia, Guopeng Zhou, and Di Liu. 2022. ArduPilot-based adaptive autopilot: Architecture and software-in-the-loop experiments. *IEEE Transactions on Aerospace and Electronic Systems* 58, 5 (2022), 4473–4485.

[7] Abraham Bernstein, Jayalath Ekanayake, and Martin Pinzger. 2007. Improving defect prediction using temporal features and non linear models. In *Proceedings of the 9th International Workshop on Principles of Software Evolution: in Conjunction with the 6th ESEC/FSE Joint Meeting*. ACM, New York, NY, 11–18.

[8] Betaflight. 2024. Betaflight Flight Controller Software (Firmware). Retrieved from https://betaflight.com/

[9] Victor Braberman, Nicolas D'Ippolito, Nir Piterman, Daniel Sykes, and Sebastian Ucriitel. 2013. Controller synthesis: From modelling to enactment. In *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*. IEEE, 1347–1350.

[10] Leo Breiman. 2001. Random forests. *Machine Learning* 45, 1 (2001), 5–32.

[11] Eric Brochu, Vlad M. Cora, and Nando De Freitas. 2010. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. arXiv:1012.2599. Retrieved from https://doi.org/10.48550/arXiv.1012.2599

[12] Jun Cai, Peng Zou, Jinxin Ma, and Jun He. 2016. SwordDTA: A dynamic taint analysis tool for software vulnerability detection. *Wuhan University Journal of Natural Sciences* 21, 1 (2016), 10–20.

[13] Darren Cofer and Steven Miller. 2014. DO-333 certification case studies. In *Proceedings of the NASA Formal Methods Symposium*. Springer, 1–15.

[14] Dario Di Nucci, Fabio Palomba, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia. 2017. A developer centered bug prediction model. *IEEE Transactions on Software Engineering* 44, 1 (2017), 5–24.

[15] Tom Erkkinen and Bill Potter. 2009. Model-based design for DO-178B with qualified tools. In *Proceedings of the AIAA Modeling and Simulation Technologies Conference and Exhibit*. American Institute of Aeronautics and Astronautics Inc. (AIAA). DOI: https://doi.org/10.2514/6.2009-6233

[16] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2019. Auto-sklearn: Efficient and robust automated machine learning. In *Automated Machine Learning*. Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (Eds.), Springer, Cham, 113–134. DOI: https://doi.org/10.1007/978-3-030-05318-5_6

[17] Antonio Filieri, Henry Hoffmann, and Martina Maggio. 2014. Automated design of self-adaptive software with control-theoretical formal guarantees. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, New York, NY, 299–310.

[18] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C. Gall. 2012. Method-level bug prediction. In *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 171–180.

[19] Emanuel Giger, Martin Pinzger, and Harald C. Gall. 2011. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, New York, NY, 83–92.

[20] Michael T. Goodrich, Roberto Tamassia, and David M. Mount. 2011. *Data Structures and Algorithms in C++*. John Wiley & Sons.

[21] Gregory A. Hall and John C. Munson. 2000. Software evolution: code delta and code churn. *Journal of Systems and Software* 54, 2 (2000), 111–118.

[22] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. 1997. HyTech: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer* 1, 1–2 (1997), 110–122.

[23] Hu Huang, Samuel Guyer, and Jason Rife. 2016. Applying machine learning for run-time bug detection in aviation software. In *Proceedings of the AIAA Infotech@ Aerospace*. AIAA Infotech@Aerospace, 0482.

[24] Daniel Jackson, Jonathan DeCastro, Soonho Kong, Dimitrios Koutentakis, A. F. Leong, Armando Solar-Lezama, Mike Wang, and Xin Zhang. 2019. Certified control for self-driving cars. In *Proceedings of the DARS 2019: 4th Workshop on the Design and Analysis of Robust Systems*. Springer. Retrieved from https://par.nsf.gov/servlets/purl/10170076

[25] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty. 2003. Model-integrated development of embedded software. *Proceedings of the IEEE* 91, 1 (2003), 145–164.
[26] Muhammad Taimoor Khan, Dimitrios Serpanos, Howard Shrobe, and Muhammad Murtaza Yousuf. 2020. Rigorous machine learning for secure and autonomous cyber physical systems. In *Proceedings of the 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA '20)*, Vol. 1. IEEE, 1815–1819.
[27] Taegyu Kim, Chung Hwan Kim, Altay Ozen, Fan Fei, Zhan Tu, Xiangyu Zhang, Xinyan Deng, Dave Jing Tian, and Dongyan Xu. 2020. From control model to program: Investigating robotic aerial vehicle accidents with {MAYDAY}. In *Proceedings of the 29th* {USENIX} Security Symposium ({USENIX} Security 20). {USENIX} Association, Online, 913–930.
[28] Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. 2019. Auto-WEKA: Automatic model selection and hyperparameter optimization in WEKA. In *Automated Machine Learning*. Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (Eds.), Springer, Cham, 81–95. DOI: https://doi.org/10.1007/978-3-030-05318-5_4
[29] Bor-Chen Kuo, Hsin-Hua Ho, Cheng-Hsuan Li, Chih-Cheng Hung, and Jin-Shiuh Taur. 2013. A kernel-based feature selection method for SVM with RBF kernel for hyperspectral image classification. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 7, 1 (2013), 317–326.
[30] Marta Kwiatkowska, Gethin Norman, and David Parker. 2007. Controller dependability analysis by probabilistic model checking. *Control Engineering Practice* 15, 11 (2007), 1427–1434.
[31] Peng Li, Di Liu, Xin Xia, and Simone Baldi. 2022. Embedding adaptive features in the ardupilot control architecture for unmanned aerial vehicles. In *Proceedings of the IEEE 61st Conference on Decision and Control (CDC '22)*. IEEE, 3773–3780.
[32] LibrePilot. 2024. LibrePilot. Retrieved from https://www.librepilot.org/
[33] Ying Ma, Guangchun Luo, Xue Zeng, and Aiguo Chen. 2012. Transfer learning for cross-company software defect prediction. *Information and Software Technology* 54, 3 (2012), 248–256.
[34] Rupak Majumdar, Indranil Saha, Koichi Ueda, and Hakan Yazarel. 2013. Compositional equivalence checking for models and code of control systems. In *Proceedings of the 52nd IEEE Conference on Decision and Control*. IEEE, 1564–1571.
[35] Reza Matinnejad, Shiva Nejati, Lionel Briand, Thomas Bruckmann, and Claude Poull. 2015. Search-based automated testing of continuous controllers: Framework, tool support, and case studies. *Information and Software Technology* 57 (2015), 705–722.
[36] Mary L. McHugh. 2012. Interrater reliability: The kappa statistic. *Biochemia Medica* 22, 3 (2012), 276–282.
[37] Yassine Mekdad, Ahmet Aris, Leonardo Babun, Abdeslam El Fergougui, Mauro Conti, Riccardo Lazzeretti, and A. Selcuk Uluagac. 2023. A survey on security and privacy issues of UAVs. *Computer Networks* 224 (2023), Article 109626. DOI: https://doi.org/10.1016/j.comnet.2023.109626
[38] Nachaat Mohamed, Abdussalam Ali Ahmed, Abdulgader Alsharif, and Hala J. ElKhozondar. 2023. Employing AI-driven drones and advanced cyber penetration tools for breakthrough criminal network surveillance. In *Proceedings of the IEEE 9th International Women in Engineering (WIE) Conference on Electrical and Computer Engineering (WIECON-ECE '23)*. IEEE, 1–6.
[39] Daniel Morariu, Radu Creţulescu, and Macarie Breazu. 2018. The weka multilayer perceptron classifier. *International Journal of Advanced Statistics and IT & C for Economics and Life Sciences* 7, 1 (2018).
[40] James Newsome and Dawn Xiaodong Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '05)*, Vol. 5. The Internet Society, 3–4. Retrieved from https://www.ndss-symposium.org/ndss2005/dynamic-taint-analysis-automatic-detection-analysis-and-signaturegeneration-exploits-commodity/
[41] PaparazziUAV. 2024. PaparazziUAV. Retrieved from https://wiki.paparazziuav.org/wiki/MainPage
[42] Nikhilkumar D. Parikh. 2016. Effectiveness of teaching through mind mapping technique. *The International Journal of Indian Psychology* 3, 3 (2016), 148–156.
[43] Brian Payne. 2014. GitHub Compare. Retrieved from https://bayne.github.io/github-compare/#!/compare/ArduPilot/ardupilot/paparazzi/paparazzi
[44] Ana Pereira and Carsten Thomas. 2020. Challenges of machine learning applied to safety-critical cyber-physical systems. *Machine Learning and Knowledge Extraction* 2, 4 (2020), 579–602.
[45] Pixhawk. 2024. Pixhawk Flight Controller Hardware Project. Retrieved from https://pixhawk.org/standards
[46] John C. Platt. 1999. Fast training of support vector machines using sequential minimal optimization, advances in kernel methods. In *Fast Training of Support Vector Machines Using Sequential Minimal Optimization, Advances in Kernel Methods*, Vol. 1. IEEE, 997–1001.
[47] Joseph Porter, Gábor Karsai, Péter Völgyesi, Harmon Nine, Peter Humke, Graham Hemingway, Ryan Thibodeaux, and János Sztipanovits. 2008. Towards model-based integration of tools and techniques for embedded control system

design, verification, and implementation. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*. Springer, 20–34.
[48] PX4. 2024. PX4 Open Source Autopilot. Retrieved from https://px4.io/
[49] Foyzur Rahman and Premkumar Devanbu. 2013. How, and why, process metrics are better. In *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*. IEEE, 432–441.
[50] Reddit. 2020. Why Enum Type Derives from uint8_t. Retrieved from https://www.reddit.com/r/cpp_questions/comments/f6cdop/why_enum_type_derives_from_uint8_t/
[51] Leanna Rierson. 2017. *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. CRC Press.
[52] Burak Turhan, Ayse Tosun, and Ayse Bener. 2011. Empirical evaluation of mixed-project defect prediction models. In *Proceedings of the 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 396–403.
[53] Frank Van Harmelen, Vladimir Lifschitz, and Bruce Porter. 2008. *Handbook of Knowledge Representation*. Elsevier, Amsterdam, The Netherlands.
[54] Kush R. Varshney and Homa Alemzadeh. 2017. On the safety of machine learning: Cyber-physical systems, decision sciences, and data products. *Big Data* 5, 3 (2017), 246–255.
[55] Amila Wickramasinghe, Nimali Widanapathirana, Osuka Kuruppu, Isurujith Liyanage, and I. M. K. Karunathilake. 2007. Effectiveness of mind maps as a learning tool for medical students. *South East Asian Journal of Medical Education* 1, 1 (2007), 30–32.
[56] Jia Wu, Xiu-Yun Chen, Hao Zhang, Li-Dong Xiong, Hang Lei, and Si-Hao Deng. 2019. Hyperparameter optimization for machine learning models based on Bayesian optimization. *Journal of Electronic Science and Technology* 17, 1 (2019), 26–40.
[57] Jean-Paul Yaacoub, Hassan Noura, Ola Salman, and Ali Chehab. 2020. Security analysis of drones systems: Attacks, limitations, and recommendations. *Internet of Things* 11 (2020), Article 100218. DOI: https://doi.org/10.1016/j.iot.2020.100218
[58] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. 2014. Towards building a universal defect prediction model. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, New York, NY, 182–191.
[59] Michael Zimmer, J. Karl Hedrick, and Edward A. Lee. 2015. Ramifications of software implementation and deployment: A case study on yaw moment controller design. In *Proceedings of the American Control Conference (ACC '15)*. IEEE, 2014–2019.
[60] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ACM, New York, NY, 91–100.

Received 11 October 2023; revised 5 May 2024; accepted 25 June 2024