Control Parameters Considered Harmful: Detecting Range Specification Bugs in Drone Configuration Modules via Learning-Guided Search

Ruidong Han
hanruidong@stu.xidian.edu.cn
Xidian University
Xian, China

Chao Yang[†]
chaoyang@xidian.edu.cn
Xidian University
Xian, China

Siqi Ma[*][†]
siqimslivia@gmail.com
The University of New South Wales
Canberra
Sydney, Australia

JiangFeng Ma
jfma@mail.xidian.edu.cn
Xidian University
Xian, China

Cong Sun
suncong@xidian.edu.cn
Xidian University
Xian, China

Juanru Li
mail@lijuanru.com
Shanghai Jiao Tong University
Shanghai, China

Elisa Bertino
bertino@purdue.edu
Purdue University
West Lafayette, USA

ABSTRACT

In order to support a variety of missions and deal with different flight environments, drone control programs typically provide configurable control parameters. However, such a flexibility introduces vulnerabilities. One such vulnerability, referred to as range specification bugs, has been recently identified. The vulnerability originates from the fact that even though each individual parameter receives a value in the recommended value range, certain combinations of parameter values may affect the drone physical stability. In this paper, we develop a novel learning-guided search system to find such combinations, that we refer to as incorrect configurations. Our system applies metaheuristic search algorithms mutating configurations to detect the configuration parameters that have values driving the drone to unstable physical states. To guide the mutations, our system leverages a machine learning based predictor as the fitness evaluator. Finally, by utilizing multi-objective optimization, our system returns the feasible ranges based on the mutation search results. Because in our system the mutations are guided by a predictor, evaluating the parameter configurations does not require realistic/simulation executions. Therefore, our system supports a comprehensive and yet efficient detection of incorrect configurations. We have carried out an experimental evaluation of our system. The evaluation results show that the system successfully reports potentially incorrect configurations, of which over 85% lead to actual unstable physical states.

KEYWORDS

Drone security, configuration test, range specification bug, deep learning approximation

ACM Reference Format:
Ruidong Han, Chao Yang[2], Siqi Ma, JiangFeng Ma, Cong Sun, Juanru Li, and Elisa Bertino. 2022. Control Parameters Considered Harmful: Detecting Range Specification Bugs in Drone Configuration Modules via Learning-Guided Search. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510084

1 INTRODUCTION

Drones – flying mini robots, are rapidly growing in popularity. They have become essential in supporting the central functions of various business sectors (e.g., motion picture filmmaking) and governmental organizations (e.g., surveillance). Due to their features of pilotless, a small physical shape, and fast speed, drones are often used in missions, such as delivery, surveillance, and targeting locations that are difficult or expensive to reach with other means. However, as drones are increasingly used for critical missions, it is essential to ensure the reliability and adaptability of the control system, towards completing missions successfully.

To achieve reliability and adaptability, enhanced flight control systems have been developed. Such systems provide large numbers of control parameters that can be configured to modify the flight states of drones, such as linear and angular positions. Through parameter adjustment, different configurations can be set and sent to the flight control program. Based on such a configuration, the flight

[*]The work was conducted while the author was lecturer at the University of Queensland.
[†]Corresponding Author

control program controls the flight states of the drone to complete flight missions. However, the functionality of adjusting parameters introduces certain vulnerabilities (referred to as range specification bugs) arising from the lack of adequate checking of the control parameter values. Specifically, when setting some particular configurations by selecting parameter values within the ranges provided by the manufacturer, unstable flight states might be triggered, such as trajectory deviation or even drone crash.

Unfortunately, existing vulnerability detection techniques cannot detect such range specification bugs. If the source of the control program is available, static program analysis can be used to detect data/control dependencies [22, 23] and find such bugs. However, such an approach works well only for small code snippets. If used on large and complex programs, static analysis would have scalability issues. To address scalability, taint analysis can be used by tracking input data flow [10, 13, 30], which relies more on input construction. However, when dealing with very large numbers of control parameters, each with a wide range of values, analyzing configurations by tainting all the parameter values is time consuming. A recently proposed tool, RVFUZZER [20], tries to address such an issue by generating configuration inputs through fuzzing. Even though RVFUZZER is able to reduce the total amount of configurations to be tested, it is still inefficient and unable to achieve high-coverage analyses of configurations. As a result, it misses incorrect configurations.

Major challenges for detecting range specification bugs include how to validate configurations effectively and how to efficiently search for correct parameter ranges. In this work, we address these challenges by developing a learning-guided fuzzing approach specifically designed to detect range specification bugs. At a high level, our detection tool, LGDFuzzer, relies on a genetic algorithm (GA) [37] and a flight state predictor to detect the configurations that are potentially incorrect and test the incorrect configuration through simulation. Specifically, LGDFuzzer is equipped with three core components, *State Change Predictor*, *Learning-Guided Mutation Searcher*, and *Range Guideline Estimator*. First, we manually collected a list of flight logs, each of which contains a flight state, sensor data, configurations, and a timestamp. By using such logs, LGDFuzzer trains a state predictor, which is utilized to estimate the state, referred to as reference state, that will be reached by the drone at the next timestamp. Simultaneously, LGDFuzzer runs the learning-guided mutation searcher to generate configurations by leveraging the GA. Unlike the traditional configuration validation schemes that test the configuration on either a flight simulator or a realistic drone, LGDFuzzer leverages the state predictor to estimate the reference state and infers whether a configuration is correct based on the deviation of the reference state concerning the expected state. Finally, LGDFuzzer validates the configurations that are predicted as "incorrect" and further generates a valid range for each parameter.

We used LGDFuzzer to analyze the most popular flight control program, *ArduPilot* [35]. In total, LGDFuzzer validated 46, 500 configurations and labeled 2, 319 as "incorrect", out of which 2, 036 incorrect configurations were confirmed. Apart from the identified range specification bugs, we also found 564 privilege escalation vulnerabilities, referred to as *Post-Launch Privilege Escalation*. These refer to configurations that are detected as incorrect before the drone takes off and, as a result, the flight is aborted. However, the control program will accept these incorrect configurations if they are sent to the control program after the drone has taken off. Our analysis also shows that a significant number of incorrect configurations are set in order to enhance adaptability. To assist developers and users in building secure flight control programs, LGDFuzzer optimizes the parameter ranges based on a manually set adaptability level. If developers and users prefer higher adaptability, larger parameter ranges will be provided by LGDFuzzer; however, the possibility of causing unstable flight would be higher. Otherwise, a smaller range will be set and the flight states of the drone will be more stable.

Contributions.

(1) We have designed and implemented LGDFuzzer to detect incorrect configurations effectively and efficiently. Our system uses a GA to select the "highly possible" incorrect configurations and validates configuration correctness by using a deep learning based state predictor.

(2) According to the requirements of reliability and adaptability by developers and users, we designed and implemented a range optimization component in LGDFuzzer to provide the most appropriate parameter value ranges to minimize the possibility of introducing incorrect configurations.

(3) We applied LGDFuzzer to a real-world flight control program and identified 2, 036 incorrect configurations causing unstable flight states. We also verified 106 incorrect configurations on real-world drones and confirmed that these incorrect configurations cause trajectory deviations or drone crashes.

(4) We found a new type of bug, Incorrect Configuration Tackling bug, and verified that this bug also causes unstable flight states.

(5) We have open sourced our LGDFuzzer at https://github.com/BlackJocker1995/uavga/tree/main; the site makes available the tool, dataset, and video recordings of our tests of incorrect configurations.

2 MOTIVATION AND CHALLENGES

In this section, we first introduce background information of flight control programs utilized by drones and range specification bugs. We then discuss the challenges in the design of an approach to validate parameter configurations, followed by our solutions to address these challenges.

2.1 Flight Control Program

During a flight, the ground control station (GCS) communicates with the drone by sending a series of commands to the flight control program. Before the drone takes off, users can configure the flight control program by adjusting the parameters to manipulate the flight states of the drone (e.g., linear position, angular position, angular speed, velocity, and acceleration). To ensure that a drone completes its flight mission successfully, the flight control program periodically observes the current positioned flight state and sensor data (e.g., from GPS, gyroscopes, and accelerometers) to estimate a reference state indicating the next state of the drone. Then the control program generates actuator signals (e.g., motor commands) to move the drone to the reference state. The positioned state and the reference state need to be close enough, i.e., within a standard deviation. If this is not the case, the drone flight may become unstable, leading to trajectory deviations and crashes.

Although the value ranges for control parameters are typically hardcoded in the control programs and one would expect that all possible combinations of these values be correct, but some of the combinations are actually incorrect. Any configurations triggering unstable flight states are regarded as incorrect. The corresponding control parameter ranges are *range specification bugs* [20]. Since hundreds of control parameters can be specified by the flight control program, identifying range specification bugs by validating all parameter values is time consuming because some parameters may not affect flight stability. Therefore, we focus on the parameters that might affect angular position and angular speed state, because they directly impact the flight attitude and their incorrect values are more likely to cause unstable states.

By analyzing the physical impact on drones, we define five unstable flight states:

Flight Freeze. A drone is required to keep moving unless the flight control program generates a signal to keep the drone stationary or to instruct the drone to land. However, incorrect configurations may lead the drone to accidentally freeze at a waypoint, when moving forward/backward, or to wander around a position within a minimum range. To determine whether a flight is frozen, we calculate the movement distance between the previous and the current positions within a time interval. If the distance is less than a threshold, the flight state is regarded as frozen, i.e.,"flight freeze".

Deviation. According to the actuator signals generated by the flight control program, the drone will be driven to achieve the reference state as close as possible. In practice, however, the positioned state may not always match the reference state, that is, the drone is deviating from the expected trajectory. When the deviation is small, the control system can generate an actuator signal based on the differences between the current positioned state and the reference state. However, an incorrect configuration may trigger a significant deviation. Such a deviation may lead to an erroneous trajectory from which the drone cannot return back to the correct trajectory. If the deviation exceeds a given threshold, we consider the flight state as unstable, i.e., a "deviation" state.

Drone Crash. For general deviations, the drone can still land safely even though not at the expected location. A worse case is a deviation leading the drone against an object and eventually to crash.

Potential Thrust Loss. By driving the drone motor, the flight control program uses motor power to adjust the drone to the reference state. Nonetheless, the adjustment that can be done by the drone motor is limited. If an incorrect configuration is set, the drone may not be able to move close to the reference state even when saturating the motor up to 100% throttle. Remaining in such an incorrect state can cause a decrease in the drone flight altitude, or even a crash.

Post-Launch Privilege Escalation. Before taking off, the flight control program validates the configuration and determines whether it will trigger unstable flight states. (i.e., the configuration is incorrect). When one or more incorrect parameters of the configurations are identified, the control program displays a warning message and aborts the taking-off operation. However, if these configurations, flagged as incorrect, are set after taking off, they can still be accepted by the flight control program. As a consequence, the drone may end up in some unstable states.

2.2 Challenges

In order to validate all configurations and detect range specification bugs, the following challenges must be addressed:

Challenge I: How to validate configurations effectively? Approaches proposed for analyzing flight control programs are generally based on static program analysis techniques that explore control and data dependencies [11, 19, 26]. However, such techniques are not suitable for validating configurations because large numbers of specified parameter values need to be analyzed. Unlike conventional bug detection techniques that statically analyze only small code snippets, the entire flight control program needs to be analyzed in order to achieve high code coverage. The reason is that different parameter values often result in execution flows involving very different portions of the control program. Unfortunately, flight control programs have huge sizes (e.g., over 700K lines of code); complicated control and data dependencies. Therefore, it is critical that the approach designed for configuration validation be effective, that is, able to provide high coverage of all possible parameter values.

Challenge II: How to conduct an efficient configuration validation? Referring to the unstable flight states defined in Section 2.1, we need to validate each configuration through either a realistic or simulated flight execution. Because of the large number of control parameters, each with a wide value range, changing the parameter values to generate configurations and validating all these configurations is inefficient. Completing the entire validation procedure may then end up requiring hours. Therefore, existing approaches [17, 32], which analyze all possible configurations, are not suitable.

An alternative approach is to use fuzzing [20] combined with a binary search [21] to reduce the search space of the combinations to be analyzed. However, although the search scope is narrowed, the execution time increases because each fuzzing iteration needs to wait until the validation feedback of the previous configuration is obtained.

Challenge III: How to balance the requirements of drone adaptability and flight stability? The flight control program supports different configurations to adapt to different flight missions. When high adaptability is required, each control parameter must have a large value range to adapt to different scenarios. As a result, the number of incorrect configurations will be higher, which will then affect flight stability. On the other hand, when parameters have small value ranges, the possible configurations are limited, which reduces adaptability. Hence, complex flight missions cannot be carried out. Identifying proper value ranges is thus challenging.

2.3 Solutions

We propose three solutions to address the above challenges.

Solution I: Grey-box based fuzzing. Since dependencies of the flight control program are complex, simple static program analysis techniques are ineffective. Our approach is instead to conduct grey-box based fuzzing by setting various configurations and validating the corresponding flight states. In particular, we apply a GA to carry out fuzzing. The algorithm first selects some parameter values and validates the parameter configuration by predicting its impact on the flight states. After that, referring to the validation result, the algorithm conducts mutation to select alternative incorrect parameter values and set new configurations to validate. The process of configuration searching is executed iteratively. This approach addresses challenges I and II.

Solution II: Flight state prediction. Although GA and fuzzing reduce the search range of parameter values, the number of combinations is still huge. Therefore, validating all the corresponding configurations through realistic/simulation execution is highly inefficient. In response, we designed a state generation approach that leverages a machine learning algorithm to train a state predictor. Instead of validating configurations through a realistic/simulation execution, the state predictor takes each configuration as input and predicts the potential flight state to guide the mutation. Such an approach requires much less time than a realistic/simulation execution. It is important to note that data labeling and predictor generation are one-time costs. Hence it is still more efficient to conduct prediction rather than a realistic/simulation execution. The reason is that configuration validation has to be executed iteratively when each new configuration is generated by the mutation algorithm. This solution addresses Challenge II.

Solution III: Multi-objective optimization. To balance adaptability and stability, we utilize a multi-objective optimization approach. Our approach estimates multiple feasible range guidelines according to the detected incorrect configurations. The optimization target is to eliminate the incorrect configurations (improve stability) while providing a wider range for parameters (improve adaptability). Each solution of the multi-objective optimization (i.e., range guideline) is the best solution in specific conditions, i.e., the optimal balance of adaptability and stability. Our approach allows users to choose an appropriate range from multiple range guidelines based on their requirements. This approach addresses Challenge III.

3 LGDFUZZER

In this section, we present the design of LGDFuzzer, a system for finding range specification bugs drone control programs through learning-guided mutation search. We first present an overview of its architecture and then the detailed design of its three components.

3.1 Overview of LGDFuzzer

LGDFuzzer (see Fig. 1) contains three components: *state change predictor*, a module to predict the flight state and assess whether a configuration will lead to unstable states; *learning-guided mutation searcher*, a GA search module to detect incorrect configurations; *range guideline estimator*, a module to provide secure parameter range guidelines.

LGDFuzzer relies on log entries that we generate by repeatedly executing drone flight missions (①). The log entries are split into

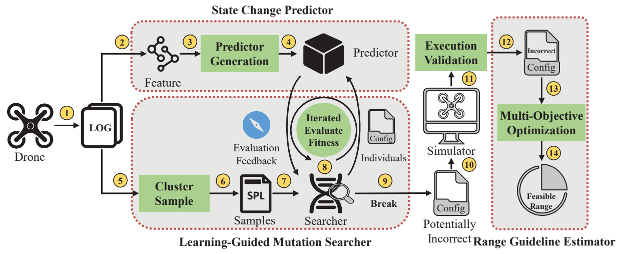

Figure 1: Overview of LGDFuzzer.

two groups, one group is used by the *state change predictor* to generate a feature matrix containing vectors with flight states, sensor data, and configurations (②), and another used by the *learning-guided mutation searcher* to carry out cluster sampling (⑤). By using the feature matrix, LGDFuzzer then generates a predictor (③④). The searcher clusters data and selects representative samples from each cluster (⑥). It further uses each representative sample to find out the corresponding incorrect configurations (⑦). In particular, LGDFuzzer iteratively mutates the configuration and uses the predictor to evaluate which configuration is more likely to cause unstable states (⑧). The iterative search will stop When certain conditions are satisfied. All the identified potential incorrect configurations are collected to construct a configuration set (⑨) and validated by a simulation (⑩⑪⑫). Finally, through the validation results, the estimator uses a multi-objective optimization to generate multiple feasible range guidelines that balance availability and stability under specific conditions (⑬⑭).

3.2 Flight Log Generation

Since there is no standard data set for testing drones, in order to construct the feature matrix for the predictor and generate data for the searcher, we manually fly a drone through a simulator to generate a number of flight logs. When flying the drone, we set each flight with the same flight test mission, *AVC2013* [34], which is often used to test the drone mission execution capabilities. The flight mission is repeatedly executed by setting configurations. We record all flight logs but discard those causing unstable states. Because unstable state data is uncontrollable and complex (compared to stable state data), dropping the unstable states prevents them from affecting the training of the predictor. In total, we recorded 308, 533 system log entries. Each entry contains state information including angular position, angular acceleration, throttle speed, sensor data of GPS, gyroscopes, and accelerometers, the current set configuration, and a timestamp index.

3.3 State Change Predictor

Since the control algorithm estimates the next reference state based on the current positioned state and sensor data, we leverage a machine learning (ML) predictor to emulate the correlation between input states and output states, and then assess how the configurations affect flight states. Through the ML predictor, LGDFuzzer explores the diversity of flight states and configurations. Then it calculates the next reference state which demonstrates the deviation caused by incorrect configuration accurately. Two steps are executed to train the predictor: *feature extraction* and *predictor generation*.

Feature Extraction. Given the log entries, LGDFuzzer constructs a feature matrix by selecting the specific data. Since we focus on the unstable states that affect flight attitude (i.e., angular and speed), the predictor considers the following state, sensor data, and configurations that might affect flight attitude: (i) angular position and angular speed of the flight state a; (ii) sensor data s obtained from gyroscopes and accelerometers; (iii) control and mission parameters of the configuration x, which could adjust flight attitude. According to the above information, LGDFuzzer groups them as a feature vector, $v\{c, x\}$, where $c = \{a, s\}$. The vectors are further combined to construct a feature matrix.

Predictor Generation. As Long Short-Term Memory (LSTM) [16] technique handles complicated input/output data efficiently [6, 7, 28], LGDFuzzer further utilizes *LSTM* as the predictor to refer the next reference state. Specifically, for a pre-processed feature vector $v_i = \{c_t, x_t\}$ with timestamp t, *LSTM* takes a number h of consecutive vectors with timestamp lower or equal than t as input (i.e., $V\{v_{t-h-1}, ..., v_{t-1}, v_t\}$), and returns the maximum conditional probability prediction of the next reference state units a'_{t+1}. We show in Sec. 4 how we determine the best values for h. In the training stage, the weight of the predictor is iteratively updated to ensure that the predicted reference state a'_{t+1} is closer to the ground truth state a_{t+1}. We use the *Mean Squared Error (MSE)* [36] loss for training.

3.4 Learning-Guided Mutation Searcher

To explore incorrect configurations, we use a GA, a metaheuristic search relying on biologically inspired operators such as mutation, crossover, and selection. Initially, as incorrect configurations may only produce unstable states in specific contexts (i.e., state and sensor data), our system conducts searches for multiple specific contexts to find incorrect configurations. We split the data from logs into multiple segments $C_i = \{c_i, c_{i+1}, ..., c_{i+h}\}$, $i \ mod \ (h+1) = 0$, where i is the index of the recorded data. But considering that there are similarities among segments, searching for each one will generate a huge number of duplicate results. To address such an issue, we cluster the segments and sample from these clusters in order to reduce redundancy while maintaining diversity. We leverage the *meanshift* [12], a probability density-based non-parametric adaptive clustering algorithm, to cluster segments and randomly sample m representative examples from each cluster for the subsequent search. Then, for each sample segment, the searcher carries out a GA search to explore incorrect configurations by iterative mutation, crossover, and selection. When the searcher has collected all the incorrect configurations for each sample, it merges and deduplicates them as a unique set, referred to as *set of potentially incorrect configurations*.

In what follows we provide details on the *Fitness Evaluation Function*, used to evaluate the fitness of a configuration, and the *Searching Process*.

Fitness Evaluation Function. The GA search applies fitness to quantify, by using the predictor, how much deviation a given configuration may cause. Intuitively, the fitness evaluation function returns the probability of deviation for a configuration. Specifically, assume that the current search is carried out for the context segment $C\{c_1, ...c_h, c_{h+1}\}$; the function selects $C'\{c_1, c_2, ...c_h\}$ to be used as input to the prediction model and a_{h+1} of c_{h+1} to be used as ground truth for calculating the deviation. When evaluating a configuration $x\{x_1, ..., x_D\}$ (D is the number of parameters), the function merges it with the segment to create features $V\{\{c_1, x\}, ..., \{c_h, x\}\}$. Then, such features are given as input to the predictor, which returns a predicted reference state a'_{h+1}. The fitness is the L1-distance $\|a_{h+1} - a'_{h+1}\|$ between the predicted state and the ground truth state. The goal of the search is then to find incorrect configurations, which are predicted to maximize the fitness, that is, the deviation (and thus maximize the probability of causing unstable states).

Searching Process. For each context segment sample, the searcher first initializes a population whose individuals are configurations, and the parameter values of each configuration are set to their default values. We assume that the population size is NP and the maximum number of iterations is G_{max}. The search process iteratively mutates and updates the population as follows.

In the g-th generation iteration ($g \in [1, G_{max}]$), the searcher first mutates the current population $pop_g\{x_{1,g}..., x_{NP,g}\}$ and generates a variant population $pop_v\{y_{1,g}, ..., y_{NP,g}\}$. Each configuration of the variant population is obtained as follows:

$$y_{i,g} = x_{i,g} + F * (x_{best,g} - x_{i,g}) + F * (x_{r1,g} - x_{r2,g}) \quad (1)$$

where $x_{r1/r2,g}$ are random configurations, $x_{best,g}$ is the best fitness configuration, and F is the scaling factor.

Then, pop_g is crossed over with pop_v to produce a new experimental population $pop_e\{e_{1,g}, ..., e_{NP,g}\}$, whose i-th configuration is $e_{i,g}\{e_{i1,g}, e_{i2,g}, ..., e_{ij,g}\}$, where $j \in [1, D]$. The parameter values $e_{ij,g}$ of each configuration are calculated as follows:

$$e_{ij,g} = \begin{cases} y_{ij,g}, & if \ rand(0,1) < CR \ or \ j = j_{rand} \\ x_{ij,g}, & otherwise \end{cases} \quad (2)$$

where j_{rand} is a random integer in $[1, D]$, CR is the crossover rate, $x_{ij,g}$ is the j-th parameter value of the i-th configuration in pop_g, and $y_{ij,g}$ is the j-th parameter value of the i-th configuration in pop_v.

Then, the searcher evaluates the fitness of each configuration both in pop_g and pop_e. According to their fitness, the searcher selects some configurations from the population to obtain the next generation of population $pop_{g+1}\{x_{1,g+1}, ..., x_{NP,g+1}\}$ by using the following selection function:

$$x_{i,g+1} = \begin{cases} e_{i,g}, & if \ f(e_{i,g}) < f(x_{i,g}) \\ x_{i,g}, & otherwise \end{cases} \quad (3)$$

where f is the fitness evaluation function.

Finally, if the fitness of each configuration in the population does not any longer increase or the maximum number G_{max} is reached, the search stops the mutation. We select the top 10 highest-fitness configurations from the final generation population as incorrect configurations.

3.5 Execution Validation

The predictor classifies a configuration as incorrect if the configuration has a high probability of causing unstable states. However, the set of all such configurations needs to be validated to confirm that they actually cause unstable states. We thus simulate the flight and use a monitor to observe which potential incorrect configurations actually lead to unstable states during the simulated execution. Specifically, the drone is set with potentially incorrect configurations to perform the *AVC2013* flight mission. For flight freeze, if the drone moves at distances always less than 0.5 meters in 15 seconds, the monitor identifies this as a flight freeze. For deviations, if the flight deviation continues to be 15 times higher than 1.5 meters, the flight is considered a deviation.

3.6 Range Guideline Estimator

The range guidelines provided to the users should consider stability and adaptability, while at the same time eliminating discrete incorrect configurations and reserving relatively complete space for each parameter. Meanwhile, as we cannot ensure the stability of unverified configurations, the estimation of range guidelines should reference the obtained validation results and refrain from considering incorrect configurations. Therefore, we leverage a multivariate optimization to determine the suitable range guideline. If all parameter values in a configuration are within the range specified by the guideline, we consider the configuration to be covered. Our system estimates the range guideline (*Range′*) based on the following optimization problem:

$$\begin{cases} min\ f_1 = \frac{num(R_{incorrect} \in Range')}{num(R \in Range')} \\ max\ f_2 = num(R \in Range') \\ s.t. \\ Range' \in OriginRange \end{cases} \tag{4}$$

The optimization problem has two objectives: (a) to maximize the number of validated configurations R covered by the guideline; (b) to minimize the coverage of incorrect configurations $R_{incorrect}$. If we shrink the allowed ranges to avoid incorrect configurations, the range of each parameter value would also shrink and vice versa. Therefore, instead of defining strict ranges for each parameter, the system solves this optimization problem with multiple constraints to obtain a diverse group of *Pareto* solutions. These *Pareto* solutions form a boundary consisting of the best feasible solutions, allowing users to select the best configuration according to their requirements. The system provides this range guideline mainly based on the following considerations: (1) As the number of parameters increases, it would be difficult to completely rule out insecure parameter values as the secure parameter ranges may not be continuous. (2) While stability is paramount, users may be willing to incur some minor risk for better controlling flexibility or availability.

4 EVALUATION

We assessed LGDFuzzer by considering its effectiveness and efficiency in validating configurations. The following research questions (**RQs**) were answered:

- **RQ1: Effectiveness.** How many incorrect configurations are detected by LGDFuzzer precisely?
- **RQ2: Adaptability.** Can LGDFuzzer provide the most suitable value ranges for parameters with minimum incorrect configurations?
- **RQ3: Enhancement.** How do the mutation and the predictor help improve configuration validation?

4.1 Experiment Setup

We applied LGDFuzzer to an open source flight control program, *ArduPilot*(4.0.3) [35], which is widely used by drone manufacturers such as *Parrot*, *Mamba*, and *Mateksys* [33]. To validate configurations specified in *ArduPilot*, we utilized three experimental vehicles (shown in Fig. 2) for testing, including two drones with *Pixhawk* [24] (i.e., *CUAV ZD550* and *AMOVLab Z410*) and a drone simulator (i.e., *Airsim* [29]).

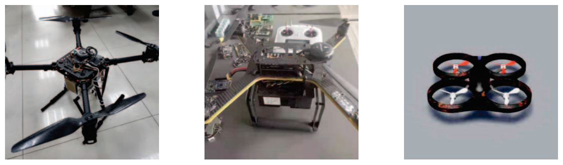
(a) CUAV ZD550 (b) AMOVLab Z410 (c) Airsim

Figure 2: Real and virtual drone vehicles used for experiments.

According to the control parameter descriptions provided by the manufacturer, we selected 20 parameters that may affect flight angular position and angular speed. We provide details about these parameters in Appendix A, including the parameter name, the parameter value range, a default value, and the parameter description.

The predictor and the GA searcher are implemented in Python. Concerning the GA, its evolutionary stagnation judgement threshold is set to 0.1, the number of representatives m is set to 3, and the maximum number of evolutionary generations is set to 200. We further set the size of initial population to $1,000$ (i.e., $NP = 1,000$) and the scaling factor F to 0.4.

4.2 RQ1: Effectiveness

To evaluate whether LGDFuzzer identifies incorrect configurations accurately, we first assessed the prediction accuracy of the predictor in the flight state predictor phase. Then we validated whether the predicted incorrect configurations can impact flight stability.

State Prediction. We first sent 1,500 configurations to *Airsim* to test and manually labeled the configurations causing an unstable state and the ones resulting in a regular state. The same method in *Fitness Evaluation Function* was used to calculate the deviation for each configuration. We calculated the average deviations of unstable states and regular states, and regarded the median values between them as a threshold. We then treated another $1,500$ configurations as test data to verify the accuracy, precision, and recall of predictors. Specifically, our experiment first extracted the context (i.e., state and sensor data) segments from log data. Then, each labeled configuration in the test data was randomly merged with a segment, and its deviation was calculated. If the deviation is greater than the threshold, the predictor classifies the configuration as unstable, and as stable otherwise. Table 1 reports the performance of predictors with different input lengths h, where **DC** is the average deviation that the correct configurations affected, and **DI** is the average deviation that incorrect configurations caused.

Table 1: Predictor accuracy, precision and recall for different input lengths.

h	Accuracy	Precision	Recall	**DC**	**DI**
3	0.8650	0.8786	0.9818	9.7643	14.6818
4	0.8689	0.8778	0.9882	9.6246	14.6508
5	0.8546	0.8777	0.9695	9.8126	14.7393
6	0.8662	0.8769	0.9553	9.6246	14.6508

* h is the input length of the model, **DC** is the average deviation of correct configurations, **DI** is the average deviation of incorrect configurations.

Generally, the results show that our predictor is robust when taking various input lengths. The deviation values derived from the incorrect configurations are obviously larger than the values collected from the correct configurations. That is the reason why we can use the predictor in the GA search to drive configurations to a higher deviation. In addition, the experiment results show that the recall of the prediction does not linearly increase when h exceeds 4. Therefore, we chose the predictor with the best recall i.e., $h = 4$ as the input length to carry out the subsequent experiments.

To assess the prediction accuracy for state changes, we sent the configurations to *Airsim* and recorded the corresponding flight data. Then, we randomly selected 150 consecutive features from the data, gave them as input to the predictor to predict the state change, and compared it with the ground truth state. The results in Fig. 3 show that the predictions closely match the real states (an example of angle roll). In the figure, the dotted curve denotes the actual states, the solid curve denotes the predicted states, and the histogram at the bottom (the blue bar) indicates the prediction errors as differences between the two curves. The small error showed that the trained predictor can accurately predict flight state changes.

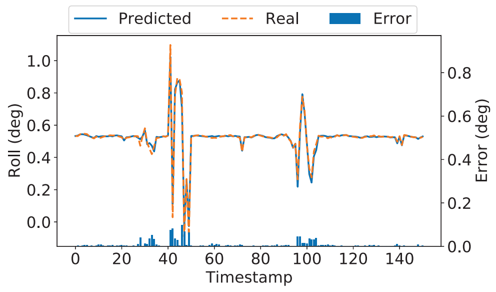

Figure 3: Match between real behavior and prediction.

Configuration Validation. Given the predicted incorrect configurations, we validated them through realistic/simulation execution. Since the incorrect configurations may cause drone crash, we examined all of them on *Airsim*. For the 465 samples obtained by clustering and sampling, LGDFuzzer searched out 2, 319 unique incorrect configurations. Finally, after a validation, 2, 036 configuration of 2, 319 were marked as truly incorrect resulting in 500 *Deviations*, 2 *Flight Freeze*, 2 *Crashes*, 564 *Post-Launch Privilege Escalation*, and 968 *Potential Thrust Losses*.

4.3 RQ2: Adaptability

In this experiment, we used the previous validation results about incorrect configurations to estimate the range guidelines, and discussed how they balance stability and adaptability. With reference to the validation results, the estimator found out 143 *Pareto* solution results (see Fig. 4). In the graph, the horizontal axis represents the number of validated configurations covered by the range guideline, and the vertical axis represents the ratio of incorrect configuration in the range guideline. Each *Pareto* solution represents a feasible solution (i.e., range guideline) satisfying specific constraints (i.e., adaptability and stability constraints).

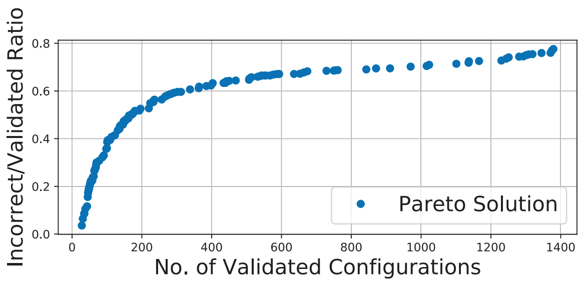

Figure 4: *Pareto* frontier solution.

For instance, we selected some range guideline examples in Table 2 and further analyzed their stability and adaptability. To illustrate the modified guidelines generated by LGDFuzzer, we selected several parameters out of 20 ones to show their details. The table shows three examples in which stability decreases and the configurable space (i.e., adaptability) increases. *Guideline* i avoids the majority of incorrect configurations. It covers 28 validated configurations, only one of which caused an unstable state; this means high stability. However, compared with the original parameter ranges, this guideline reduces too much the available space, which resulted in low adaptability. In contrast, *Guideline* iii reserves relatively complete ranges for the parameters. It covers 236 validated configurations but more than half of them (133) are incorrect, which resulted in low stability. *Guideline* ii is an intermediate choice, covering 91 validated configurations where only 29 are incorrect.

If users have more stringent stability requirements, they can use the lower error ratio range guideline, at the cost of limiting the configuration space and thus being unable to satisfy other flight requirements. On the contrary, if users have to satisfy special mission requirements (e.g., the mission is a time-limited task, or it needs a large flight angle to reach the target speed), they may consider sacrificing a bit the stability in order to improve adaptability. In fact, they can choose an appropriate range guideline from the *Pareto* solutions according to their stability and adaptability requirements.

Table 2: Examples of feasible range guideline

Parameter	*Guideline* i			*Guideline* ii			*Guideline* iii		
	L	**U**	Reduce	**L**	**U**	Reduce	**L**	**U**	Reduce
PSC_POSZ_P	1.00	2.57	-21.5%	1.00	2.59	-20.5%	1.00	2.97	-1.5%
ATC_ANG_RLL_P	0.92	11.4	-12.6%	0.88	11.73	-9.5%	1.02	11.69	-11.0%
ATC_ANG_PIT_P	1.22	11.69	-12.7%	0.94	11.78	-9.6%	1.00	11.93	-8.9%
ATC_RAT_PIT_P	0.035	0.39	-27.5%	0.035	0.375	-30.6%	0.01	0.36	-28.5%
ATC_RAT_PIT_I	0.13	1.92	-10.0%	0.11	1.93	-8.5%	0.10	1.96	-6.5%
ATC_RAT_YAW_I	0.01	0.69	-31.3%	0.01	0.72	-28.2%	0.01	0.95	-5.0%
WPNAV_SPEED	250	1550	-34.3%	250	1850	-19.1%	100	1800	-14.1%
WPNAV_SPEED_UP	150	650	-49.4%	50	650	-39.3%	50	650	-39.3%
I,**C**,**V**	1,27,28			29,62,91			133,103,236		
Coverage(covered/total validation)	0.05%,9.5%,1.2%			1.4%,21.9%,3.9%			6.5%,36.3%,10.1%		

* **L** is range lower bound, **U** is range upper bound, reduced is calculated relative to original range, **I** is the number of incorrect configurations covered by the range guidance, **C** is the number of correct configurations covered by the range guidance, **V** is the number of validated configurations covered by the range guidance.

4.4 RQ3: Improvement

To show the advantages of our system, we experimentally compared it with RVFUZZER [20], a recent approach for searching range specification bug. RVFUZZER relies on the *One-dimensional Mutation* and *Multi-dimensional Mutation* search to detect incorrect configurations. By using *One-dimensional Mutation*, centered on the default parameter values, RVFUZZER separately conducts binary searches to narrow the upper and lower bounds until a midpoint value is obtained that does not any longer cause unstable states. In the *Multi-dimensional Mutation* multiple parameters are considered, each of which determines a novel binary mutation search configured with different extreme values (i.e., maximum and minimum) of the other parameters. All these experiments were based on the six parameters utilized in RVFUZZER (see Appendix A). In the experiments, we considered the unstable states listed in Section 2.1.

Comparison with Respect to Missed Incorrect Configurations. In the *One-dimensional Mutation*, the optimal solution may not be consistent with the right optimum. For example, using such a mutation strategy, the search indicated that the correct range for `INS_POS1_Z` should remain within $[-4.7, 0.0]$ when it searched for the lower bound. However, there were still incorrect configurations, specifically between -1.0 and 0.0, inside this range. The reason is that, because of the binary search, as the first midpoint (i.e., -2.5) did not cause an unstable state, the search directly skipped values greater than -2.5. It thus did not cover the $[-1.0, 0.0]$ space, which resulted in missing potentially incorrect configurations.

In the case of multiple parameters mutations, we first applied the *Multi-dimensional Mutation* to determine the correct range. After that, based on this correct range, we leveraged LGDFUZZER to start another search to evaluate whether there are incorrect configurations. LGDFUZZER still detected 727 potentially incorrect configurations, of which only 185 were confirmed. Such a result indicated that the ranges provided by the *Multi-dimensional Mutation* are not correct. In our opinion, the *Multi-dimensional Mutation* is still a one-dimension mutation, because it uses binary search to mutate parameters separately but only imports the extremes of the value ranges of other parameters. It can be regarded as multiple one-dimensional mutations with a limited correlation between control parameters; as such, it does not consider the influence of values different from the extremes of the ranges.

Comparison with Respect to Correct Range Guidelines. The six parameters utilized in this experiment are listed in Appendix A. We leveraged LGDFUZZER to search incorrect configurations for these six parameters. The search detected 1,199 unique potentially incorrect configurations. Then the validation determined that 714 out of those configurations actually leaded to unstable states. After that, we used them to generate the feasible range guidelines and chose the lowest incorrect ratio result. Table 3 lists the ranges obtained by three methods, where *1* is *One-dimensional mutation*, *M* is *Multi-dimensional mutation*, and *GA* (our *Genetic Algorithm mutation*).

RVFUZZER method *1* obtained little reduction for each parameter range; thus it is not able to rule out incorrect configurations. *M* avoided some of the incorrect configurations but still misses others. Because our *GA* greatly reduced the ranges, it provided high stability. A special case is related to `INS_POS3_Z`; the range given by *GA* is larger than the range given by *M*. But this does not mean that our range for `INS_POS3_Z` is incorrect. The reason is that, as other parameters have a smaller range, `INS_POS3_Z` can have a larger range, since the validity of configurations is decided based on multiple parameters instead of a single one. As for other parameters, `ANGLE_MAX` influences the flight inclination angle. Under the influence of other parameters, a too large value for `ANGLE_MAX` is more likely to cause flight problems. A too small value for the waypoint speed `WPNAV_SPEED` makes the drone more likely to freeze; both *M* and *GA* reduces the lower part of the range for this parameter. For `INS_POS*_Z`, the ranges returned by *GA* are smaller than the ones returned by *M*, and closer to default values. In fact, changing `INS_POS*_Z` influences the position judgment of the inertial measurement unit. Therefore, this parameter should be changed carefully and should not deviate too much from the default value. `PSC_VELXY_P` affected the output gain of the system for acceleration; a too large gain can easily cause drone deviations or thrust losses.

Table 3: Comparison of range guideline.

Parameter	*1*		*M*		*GA*	
	L	**U**	**L**	**U**	**L**	**U**
PSC_VELXY_P	0.1	6.0	0.6	6.0	1.9	4.2
INS_POS1_Z	-4.7	5.0	-1	4.1	-0.5	2.1
INS_POS2_Z	-5.0	5.0	-0.7	3.2	-0.7	1.2
INS_POS3_Z	-5.0	5.0	-0.8	3.0	-0.9	3.2
WPNAV_SPEED	50	2000	300	2000	50	1950
ANGLE_MAX	1000	8000	1000	8000	1100	4650
I,**V**,**C**	699/476/1175		32/18/50		0/7/7	

* *1*: One-dimensional mutation, *M*: Multi-dimensional mutation, *GA*: Genetic Algorithm mutation, **L** is range lower bound, **U** is range upper bound, **I** is the number of incorrect configurations covered by the range guidance, **C** is the number of correct configurations covered by the range guidance, **V** is the number of validated configurations covered by the range guidance.

Comparison with Respect to Time Requirements. We analyzed the time taken by LGDFuzzer and *Multi-dimensional Mutation*. To determine the range guideline for the six parameters, we started multiple simulations and took 696 seconds to collect log data. The predictor took 700 seconds to train until reaching convergence. The GA search took another 156 seconds to iterate and updated its population (1, 000 configurations). Finally, from generating data to searching out 1, 000 incorrect configurations, our system totally took 1, 552 seconds. Also, over 85% configurations were validated and detected to actually cause unstable states. On the contrary, depending on the configuration values, rvfuzzer usually took between 20 and 120 seconds per round to complete a *AVC2013* flight mission. Even when we allocated 1, 552 seconds to *Multi-dimensional Mutation*, it can only carry out 78 rounds of mission test. In fact, the *Multi-dimensional Mutation* search took roughly 2 hours to estimate the guidelines for the six parameters. In addition, for 20 parameters, the time taken by *Multi-dimensional Mutation* increased exponentially. Instead, the time consumption of LGDFuzzer was still closer to the time consumption required by six parameters because the time required by predictor was almost unchanged.

4.5 Case Studies

After obtaining the fuzzing results, we selected several representative examples of unstable states to analyze their characteristics.

Deviation. There are two main deviation situations in our experiment: overshoot and fly away. When flying from one waypoint to another, the drone first accelerates to reach the target velocity and decelerates while approaching the next waypoint. If the configuration is set improperly, the drone is unable to reduce its speed when close to the waypoint, which caused an overshoot deviation (see the video of a simulation in which the drone is not able to break at [3]). In comparison, fly away is much more dangerous, in that the drone keeps moving away from its mission-planned path. We leveraged a real drone experiment to demonstrate the damage resulting from those unstable states. The experiment set up a simple surround flight mission and a drone configuration is set that, in the simulation test, resulted in a fly away deviation. As shown in the video [2], the drone deviated from the mission-planned path. Unfortunately, even though we used the RC controller (remote manual controller) to manually switch its mode to *land* in order to stabilize the drone and make sure it would land slowly, the drone was still unable to stabilize and kept deviating. In fact, after checking the offline flight log, when we manually switched to the *land* mode, the system reported a switching error indicating that it could not stabilize and land. In other words, if the incorrect configuration is not dealt with in time, the flight stability cannot be even corrected manually.

Flight Freeze. There are two main situations of flight freeze. On the one hand, an incorrect control gain parameter makes the drone fail to reach the desired position in time, or prolong the time to reach a particular waypoint or the desired position. On the other hand, an invalid configuration affects the position and causes the drone to around-flight within a small area. We tested the around-flight situation with a real drone experience. The video at [4] showed that the drone keeps circling and was unable to complete the given mission. We analyzed the offline flight log, and from the log, we could see that the drone kept flight switching between *althold* (hovering fly) and *land* but could not land successfully.

Drone Crash. Drone crash may be caused by a rollover during takeoff or by hitting the ground due to improper descent speed. The video at [1] showed an actual example of a drone crashing when taking off.

Potential Thrust Loss. This situation is mainly caused by an excessive change in angle or speed; the flight controller attempts to recover its position to normal, but the power of the motor cannot satisfy the requirement. Failure in recovering from the thrust loss or stabilizing the altitude would lead to a drone crash. In the postmortem analysis, we found that changing the control parameters related to the inertial measurement unit position (e.g., `INS_POS*_Z`) and the PID angle controller (e.g., `ATC_ANG_*_P/I/D`) can force the drone into a gradually amplified oscillation.

Post-Launch Privilege Escalation. Post-Launch privilege escalation is the new type of error we defined. A control program usually contains a checking mechanism to prevent obvious errors in configurations. If the parameters are related to the position controller (e.g., `ATC_*_*_P/I/D`), the incorrect configuration will raise a warning when the GCS tries to arm the drone. But in fact, if we update the configuration after the drone takes off, the problem is that the control system still accepts it, which ultimately drives the drone to become unstable and out of control. The video at [5] showed an example in which uploading an incorrect configuration during the mission causes the drone to crash.

5 RELATED WORK

5.1 Drone Fault Detection

A large number of drone fault detection methods/systems have been proposed to prevent errors during flights. Among them, an invariant approach [14] uses the control invariant to identify physical attacks against robotic vehicles. But their approach is unable to prevent attacks that exploit the range specification bug. To detect faulty components in a drone system, a fault detection method [18] constructs an observer model to estimate the output in fault-free conditions from the history inputs and outputs. Such a system utilizes the difference between actual outputs and the predicted values to detect faulty sensor and actuator components, but it still does not consider the instability caused by range specification bug. A neural network based validation approach [27] leverages the analytical redundancy between flight parameters to detect sensor faults. Like the other approaches, it only considers external factors. In comparison, LGDFuzzer implements a search system to avoid incorrect configurations that are instability factors within the system.

5.2 Learning-Based Testing

From the perspective of learning-based testing, there are three relevant systems or methods. NEUZZ [31] is a gradient-guided search strategy. It uses a feed-forward neural network to approximate program branching behaviors and predict the control flow edges to cover more test space, but the predictive model is not used for guiding testing. ExploitMeter [38] uses dynamic fuzzing tests with machine learning-based prediction to update the belief in software exploitability. A learning-guided fuzzing [9] uses an LSTM network to model the input-output relation of the target system and a meta-heuristic method to search for specific actuator commands that could drive the system into unsafe states. A reinforcement fuzzing [8] applies a deep Q-learning [25] algorithm to generate an optimized policy for the fuzzing-based testing of PDF processing programs. DeepSmith [15] trains a generative model with a large corpus of open source code and uses this model to produce testing inputs to examine the OpenCL compiler automatically. These approaches make use of prior knowledge to drive the mutation input. Similarly, we introduced a machine learning model to guide the search test process. However, our LGDFuzzer combines the model with the genetic algorithm to carry out a large-scale multi-parameter search.

6 CONCLUSION

Incorrect configurations of drone control parameters, set by legitimate users or worst sent by attackers, can result in flight instabilities disrupting drone missions. In this paper, we propose a fuzzing-based system that efficiently and effectively detects the incorrect parameter configurations. LGDFuzzer uses a machine learning-guided fuzzing approach that uses a predictor, a genetic search algorithm, and multi-objective optimization to detect incorrect configurations and provide correct feasible ranges. We have experimentally compared LGDFuzzer with a state-of-the art tool. The experimental results show that LGDFuzzer is superior to such a tool in all respects. Even though our methodology has been designed for aerial drones, we believe that it can be used for other mobile devices with complex controls, such as underwater drones.

ACKNOWLEDGMENT

We would like to thank anonymous reviewers for the insightful comments. This work is supported by the Foundation for Innovative Research Groups of the National Natural Science Foundation of China (Grant No. 62121001).

REFERENCES

[1] Demo Video 2021. [n.d.]. Crash. https://youtu.be/TZFcyl5d2mk.

[2] Demo Video 2021. [n.d.]. Deviation with flying away . https://youtu.be/0buFPNhkLJc.

[3] Demo Video 2021. [n.d.]. Deviation with overshoot. https://youtu.be/imsBGaX-8ug.

[4] Demo Video 2021. [n.d.]. Flight Freeze. https://youtu.be/g79lsTp6H4g.

[5] Demo Video 2021. [n.d.]. Post-Launch Privilege Escalation Leads to Crash. https://youtu.be/B_WULGY-Dbg.

[6] Ryo Akita, Akira Yoshihara, Takashi Matsubara, and Kuniaki Uehara. 2016. Deep learning for stock prediction using numerical and textual information. In *Proceedings of the 15th IEEE/ACIS International Conference on Computer and Information Science (ICIS)*. IEEE, 1–6.

[7] Florent Altché and Arnaud de La Fortelle. 2017. An LSTM network for highway trajectory prediction. In *Proceedings of the 20th IEEE International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 353–359.

[8] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. 2018. Deep reinforcement fuzzing. In *Proceedings of the 2018 IEEE Security and Privacy Workshops (SPW)*. IEEE, 116–122.

[9] Yuqi Chen, Christopher M Poskitt, Jun Sun, Sridhar Adepu, and Fan Zhang. 2019. Learning-Guided network fuzzing for testing cyber-physical system defences. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 962–973.

[10] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. 2018. DTaint: detecting the taint-style vulnerability in embedded device firmware. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 430–441.

[11] Long Cheng, Ke Tian, Danfeng Daphne Yao, Lui Sha, and Raheem A. Beyah. 2021. Checking is Believing: Event-Aware program anomaly detection in cyber-physical systems. *IEEE Transactions on Dependable and Secure Computing* 18, 2 (2021), 825–842.

[12] Yizong Cheng. 1995. Mean shift, mode seeking, and clustering. *IEEE transactions on pattern analysis and machine intelligence* 17, 8 (1995), 790–799.

[13] Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin Vechev. 2019. Scalable taint specification inference with big code. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 760–774.

[14] Hongjun Choi, Wen-Chuan Lee, Yousra Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, Dongyan Xu, and Xinyan Deng. 2018. Detecting attacks against robotic vehicles: A control invariant approach. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 801–816.

[15] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 95–105.

[16] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. 2000. Learning to forget: Continual prediction with LSTM. *Neural Computation* 12, 10 (2000), 2451–2471.

[17] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2019. Test them all, is it worth it? Assessing configuration sampling on the jHipster web development stack. *Empirical Software Engineering* 24, 2 (2019), 674–717.

[18] G Heredia, A Ollero, M Bejar, and R Mahtani. 2008. Sensor and actuator fault detection in small autonomous helicopters. *Mechatronics* 18, 2 (2008), 90–99.

[19] 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)*. USENIX Association, 913–930.

[20] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. 2019. RVFuzzer: Finding input validation bugs in robotic vehicles through control-guided testing. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*. USENIX Association, 425–442.

[21] Donald E. Knuth. 1971. Optimum Binary Search Trees. *Acta informatica* 1, 1 (1971), 14–25.

[22] Siqi Ma, Juanru Li, Hyoungshick Kim, Elisa Bertino, Surya Nepal, Diethelm Ostry, and Cong Sun. 2021. Fine with "1234"? An analysis of SMS one-Time password randomness in android apps. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 1671–1682.

[23] Siqi Ma, Juanru Li, Surya Nepal, Diet Ostry, David Lo, Sanjay Jha, Robert H Deng, and Elisa Bertino. 2021. Orchestration or automation: authentication flaw detection in android apps. *IEEE Transactions on Dependable and Secure Computing* (2021).

[24] Lorenz Meier, Petri Tanskanen, Friedrich Fraundorfer, and Marc Pollefeys. 2011. Pixhawk: A system for autonomous flight using onboard computer vision. In *Proceedings of the 2011 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2992–2997.

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[26] Xiao rui Zhu, Chen Liang, Zhen guo Yin, Zhong Shao, Meng qi Liu, and Hao Chen. 2019. A new hierarchical software architecture towards safety-critical aspects of a drone system. arXiv:1905.06768 [cs.SE]

[27] Ihab Samy, Ian Postlethwaite, and Dawei Gu. 2008. Neural network based sensor validation scheme demonstrated on an unmanned air vehicle (UAV) model. In *Proceedings of the 47th IEEE Conference on Decision and Control (CDC)*. IEEE, 1237–1242.

[28] Sreelekshmy Selvin, R Vinayakumar, E. A Gopalakrishnan, Vijay Krishna Menon, and K. P. Soman. 2017. Stock price prediction using LSTM, RNN and CNN-sliding window model. In *Proceedings of the 2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, 1643–1647.

[29] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. 2017. AirSim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*. arXiv:arXiv:1705.05065

[30] Dongdong She, Yizheng Chen, Abhishek Shah, Baishakhi Ray, and Suman Jana. 2020. Neutaint: Efficient dynamic taint analysis with neural networks. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1527–1543.

[31] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient fuzzing with neural program smoothing. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 803–817.

[32] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. 2011. Configuration coverage in the analysis of large-scale system software. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS)*. ACM, Article 2, 5 pages.

[33] ArduPilot Team. [n.d.]. Autopilot Hardware Options. https://ardupilot.org/copter/docs/common-autopilots.html#closed-hardware.

[34] ArduPilot Team. [n.d.]. SparkFun autonomous vehicle competition 2013. https://avc.sparkfun.com/2013.

[35] ArduPilot Dev Team. [n.d.]. Ardupilot - Versatile, Trusted, Open Autopilot software for drones and other autonomous systems. https://ardupilot.org/.

[36] Zhou Wang and Alan C. Bovik. 2009. Mean squared error: Love it or leave it? A new look at Signal Fidelity Measures. *IEEE Signal Processing Magazine* 26, 1 (2009), 98–117.

[37] Darrell Whitley. 1994. A genetic algorithm tutorial. *Statistics and computing* 4, 2 (1994), 65–85.

[38] Guanhua Yan, Junchen Lu, Zhan Shu, and Yunus Kucuk. 2017. Exploitmeter: Combining fuzzing with machine learning for automated evaluation of software exploitability. In *Proceedings of the 2017 IEEE Symposium on Privacy-Aware Computing (PAC)*. IEEE, 164–175.

A DESCRIPTION OF PARAMETERS

Table 4: Parameters of control program for experiments.

Control Module	Parameter	Range	Default	Description.
Controller	PSC_POSXY_P	[0.50, 2.00]	1.0	Position controller P gain. Converts the distance (in the latitude direction) to the target location into a desired speed which is then passed to the loiter latitude rate controller.
	PSC_VELXY_P	[0.10, 6.00]	2.0	Velocity (horizontal) P gain. Converts the difference between desired velocity to a target acceleration.
	PSC_POSZ_P	[1.00, 3.00]	1.0	Position (vertical) controller P gain. Converts the difference between the desired altitude and actual altitude into a climb or descent rate which is passed to the throttle rate controller.
	ATC_ANG_RLL_P	[0.00, 12.0]	4.5	Roll axis angle controller P gain. Converts the error between the desired roll angle and actual angle to a desired roll rate.
	ATC_RAT_RLL_I	[0.01, 2.00]	0.135	Roll axis rate controller I gain. Corrects long-term difference in desired roll rate vs actual roll rate.
	ATC_RAT_RLL_D	[0.00, 0.05]	0.0036	Roll axis rate controller D gain. Compensates for short-term change in desired roll rate vs actual roll rate.
	ATC_RAT_RLL_P	[0.01, 0.50]	0.135	Roll axis rate controller P gain. Converts the difference between desired roll rate and actual roll rate into a motor speed output.
	ATC_ANG_PIT_P	[0.00, 12.0]	4.5	Pitch axis angle controller P gain. Converts the error between the desired pitch angle and actual angle to a desired pitch rate.
	ATC_RAT_PIT_P	[0.01, 0.50]	0.135	Pitch axis rate controller P gain. Converts the difference between desired pitch rate and actual pitch rate into a motor speed output.
	ATC_RAT_PIT_I	[0.01, 2.00]	0.135	Pitch axis rate controller I gain. Corrects long-term difference in desired pitch rate vs actual pitch rate.
	ATC_RAT_PIT_D	[0.00, 0.05]	0.0036	Pitch axis rate controller D gain. Compensates for short-term change in desired pitch rate vs actual pitch rate.
	ATC_ANG_YAW_P	[0.00, 6.00]	4.5	Yaw axis angle controller P gain. Converts the error between the desired yaw angle and actual angle to a desired yaw rate.
	ATC_RAT_YAW_P	[0.10, 2.50]	0.18	Yaw axis rate controller P gain. Converts the difference between desired yaw rate and actual yaw rate into a motor speed output.
	ATC_RAT_YAW_I	[0.01, 1.00]	0.018	Yaw axis rate controller I gain. Corrects long-term difference in desired yaw rate vs actual yaw rate.
	ATC_RAT_YAW_D	[0.00, 0.02]	0	Yaw axis rate controller D gain. Compensates for short-term change in desired yaw rate vs actual yaw rate.
Mission	WPNAV_SPEED	[20, 2000]	500	Defines the speed in cm/s which the aircraft will attempt to maintain horizontally during a Waypoint mission.
	WPNAV_SPEED_DN	[10, 500]	150	Defines the speed in cm/s which the aircraft will attempt to maintain while descending during a Waypoint mission.
	WPNAV_SPEED_UP	[10, 1000]	250	Defines the speed in cm/s which the aircraft will attempt to maintain while climbing during a Waypoint mission.
	WPNAV_ACCEL	[50, 500]	100	Defines the horizontal acceleration in cm/s^2 used during missions.
	ANGLE_MAX	[1000, 8000]	4500	Maximum lean angle in all flight modes.

Table 5: Parameters of control program for comparisons.

Control Module	Parameter	Range	Default	Description.
Controller	PSC_VELXY_P	$[0.0, 6.0]$	2.0	Converts the difference between desired velocity to a target acceleration.
Sensor	INS_POS1_Z	$[-5.0, 5.0]$	0.0	Z position of the first IMU accelerometer in body frame.
	INS_POS2_Z	$[-5.0, 5.0]$	0.0	X position of the second IMU accelerometer in body frame.
	INS_POS3_Z	$[-5.0, 5.0]$	0.0	Y position of the second IMU accelerometer in body frame.
Mission	WPNAV_SPEED	[20, 2000]	500	Defines the speed in cm/s which the aircraft will attempt to maintain horizontally during a Waypoint mission.
	ANGLE_MAX	[1000, 8000]	4500	Maximum lean angle in all flight modes.