Range Specification Bug Detection in Flight Control System Through Fuzzing

Ruidong Han, Siqi Ma, *Member, IEEE*, Juanru Li, *Member, IEEE*, Surya Nepal, *Senior Member, IEEE*, David Lo, *Fellow, IEEE*, Zhuo Ma, *Member, IEEE*, and JianFeng Ma, *Member, IEEE*

***Abstract*—Developers and manufacturers provide configurable control parameters for flight control programs to support various environments and missions, along with suggested ranges for these parameters to ensure flight safety. However, this flexible mechanism can also introduce a vulnerability known as range specification bugs. The vulnerability originates from the evidence that certain combinations of parameter values may affect the drone's physical stability even though its parameters are within the suggested range. The paper introduces a novel system called ICSEARCHER, designed to identify incorrect configurations or unreasonable combinations of parameters and suggest more reasonable ranges for these parameters. ICSEARCHER applies a metaheuristic search algorithm to find configurations with a high probability of driving the drone into unstable states. In particular, ICSEARCHER adopts a machine learning-based predictor to assist the searcher in evaluating the fitness of configuration. Finally, leveraging searched incorrect configurations, ICSEARCHER can summarize the feasible ranges through multi-objective optimization. ICSEARCHER applies a predictor to guide the search, which eliminates the need for realistic/simulation executions when evaluating configurations and further promotes search efficiency. We have carried out experimental evaluations of ICSEARCHER in different control programs. The evaluation results show that the system successfully reports potentially incorrect configurations, of which over 94% leads to unstable states.**

***Index Terms*—Drone security, configuration test, range specification bug, deep learning approximation.**

Manuscript received 10 October 2023; revised 5 January 2024; accepted 12 January 2024. Date of publication 17 January 2024; date of current version 15 March 2024. This work was supported in part by the National Natural Science Foundation of China (Key Program) under Grant 62232013, in part by the Major Research Plan of the National Natural Science Foundation of China under Grant 92267204 and Grant 92167203, and in part by the National Natural Science Foundation of China under Grants 62302363. Recommended for acceptance by D. Bianculli. *(Corresponding author: Ruidong Han.)*

Ruidong Han is with the School of Cyber Engineering, Xidian University, Xi'an 710071, China, and also with the School of Computing and Information Systems, Singapore Management University, Singapore 188065 (e-mail: ruidonghanxd@outlook.com).

Siqi Ma is with the School of Engineering and Information System, University of New South Wales, Sydney, NSW 2052, Australia.

Juanru Li is with Zhiyuan College, Shanghai Jiao Tong University, Shanghai 200240, China.

Surya Nepal is with the Commonwealth Scientific and Industrial Research, Sydney, NSW 2007, Australia.

David Lo is with the School of Computing and Information Systems, Singapore Management University, Singapore 188065.

Zhuo Ma and JianFeng Ma are with the School of Cyber Engineering, Xidian University, Xi'an 710071, China.

Digital Object Identifier 10.1109/TSE.2024.3354739

I. Introduction

Advances in electronics and sensor technology widen the scope of drone applications [1], [2] in traffic monitoring, field rescue, commercial courier, and remote sensing. Since flight scenarios are complex and uncontrollable, flight control programs (e.g., *PX4* [3], *LibrePilot* [4], and *ArduPilot* [5]) implement an adjustment mechanism, parameter configuration, to ensure the reliability (keep stable flight) and adaptability (suitable for more scenarios) of the flight control. Such a mechanism provides large numbers of parameters to adapt discrepancies of flight hardware and mission, like flight dynamics model and maximum flight inclination. A reliable and flexible control program encompasses hundreds of flight parameters influencing the behavior of flights.

However, the safety of the configuration is unpredictable unless a real flight test is executed, which makes the inappropriate configuration possible to upload to the drone. These configurations could cause the drone to make an irreversibly unstable flight. For instance, if a flight deviates from its planned path, leading to trajectory deviation; the actuator can not provide more power to maintain flight, causing thrust loss and even a flight crash. To maintain the safety and reliability of a configuration, developers or manufacturers of control programs provide suggestion ranges for parameter value choices to prevent unpredictable events like these unstable flights. While developers paved the way in uploading a suitable and safe configuration, it still exposes certain vulnerabilities (range specification bugs) as a lack of adequate checking of parameter values. Specifically, even if some particular configurations whose parameter values are within the ranges suggestion, these unstable flights might still be triggered; that is, the official suggestion can not prevent configurations from driving the drone into an unstable flight.

Unfortunately, existing vulnerability detection techniques for drones do not aim to detect/search for such range specification bugs. Taint analysis [6], [7], [8], [9] track the parameter data flow to locate its impact on the system. However, when dealing with very large numbers of control parameters, each with a wide range of values, tainting all values is time-consuming. In addition, this tracking is available while the program is running, which means that the localization of the problem requires the drone to be "dangerous". A recently proposed approach, RVFUZZER [10], applies a fuzzing test to address such an issue. Even though RVFUZZER proposes two binary search methods to fuzzing test the number of configurations, it is still unable to achieve high coverage and misses some search spaces containing incorrect configurations. LGDFUZZER [11] adopts a fuzzing with a learning model to evaluate configuration and accelerate the searching process. However, the evaluation in that work only considers a predicted deviation in one timestamp, which search results are likely to be affected by transient disturbances, affecting the accuracy of the search.

The major challenges of range specification bug detection are: 1) the verification of configurations requires execution results, which enormously reduces the efficiency of the search process; 2) the curse of dimensionality and the complex correlation among parameters further intensify the difficulty of covering all possible parameter combinations; 3) transient errors or data noise may affect search judgments; 4) as range adaptability and flight stability conflict (e.g., a wider configurable range is more likely to cause unstable flight, more stable otherwise), summarizing appropriate ranges to balance range adaptability and flight stability is difficult.

This work addresses the challenges by developing a fuzzing approach with a predictor to search for incorrect configurations and provide guidance. At a high level, our approach, ICSEARCHER, relies on a genetic algorithm (GA) [12] and a flight state predictor to detect the potentially incorrect configurations. Specifically, the approach consists of three modules: *reference state predictor*, *incorrect configuration searcher*, and *flexible range guide*. First, we manually collected flight logs, each entry containing flight status, sensor data, configuration, and timestamps. ICSEARCHER then generates a predictor that is applied to estimate a segment of further reference flight states. Simultaneously, ICSEARCHER carries out a GA searcher to find potential incorrect configurations that will lead to unstable assessed by the predictor. Unlike the traditional fuzzing validation schemes that test a candidate through a realistic or simulation execution, ICSEARCHER substitutes the feedback evaluation in GA with that predictor, using probabilistic prediction results to drive the search. Finally, ICSEARCHER validates the configurations predicted as "incorrect" and further generates flexible ranges for parameters.

We used ICSEARCHER to analyze two popular flight control programs, *ArduPilot* and *PX4*. For *ArduPilot*, ICSEARCHER raised $4,386$ potential incorrect configurations, out of which $4,157$ were confirmed. For *PX4*, ICSEARCHER raised $2,282$ potential incorrect configurations, out of which $2,087$ were confirmed. Our analysis also shows that many incorrect configurations are set to enhance adaptability. ICSEARCHER provides parameter ranges based on manually set adaptability levels. If developers and users prefer more adaptability, ICSEARCHER provides a larger parameter range; however, it has a higher probability of causing unstable states. Otherwise, a smaller range will be set, and the flying state of the drone will be more stable.

Contributions.

- We designed and implemented ICSEARCHER to search incorrect configurations effectively and efficiently. The system applies a GA to search "highly probability" incorrect configurations, which validates configurations through a deep learning-based predictor.

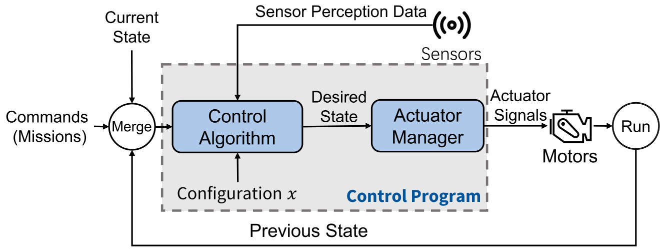


Fig. 1. Control logic of control program.

- We use segment deviation data to quantify the effect of configuration to mitigate the impact of transient bias or noise.
- To balance the reliability and adaptability for developers' requirements, our system leverages an optimization approach to provide multiple flexible range guidelines to minimize the possibility of including incorrect configurations while keeping a wide range space.
- We applied ICSEARCHER to a real-world flight control program and identified $4,386$ for *Ardupilot* and $2,087$ for *PX4* 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.
- We have open sourced our ICSEARCHER at https://github.com/BlackJocker1995/uavga/tree/main and https://figshare.com/articles/software/Source_code_of_ICSearcher_/24947442; the site makes available the tool.

II. BACKGROUNDS

We introduce the background of drone flight control programs and range specification bugs.

A. Flight Control Program

The flight control program is a framework containing three main parts: vehicle-specific flight code, shared libraries, and a hardware abstraction layer. Specifically, vehicle-specific flight code defines the firmware of drones, which processes the received command and calculates the following flight status of the drone. Shared libraries include sensor drivers, attitude and position estimation (aka EKF), and control code, and the hardware abstraction layer makes the flight control program portable to different platforms.

Fig. 1 demonstrates the control process of the program. Given a command, the flight control program predicts the next desired state (i.e., *reference state*) according to the previous states, current state, sensor perception data collected from various sensors (e.g., GPS, gyroscopes, and accelerometers), and flight configuration given by users. It then generates actuator signals (e.g., motor commands) to drive the drone to the reference state. A drone is considered stable if the actual state is close enough to the reference state within a standard deviation. Otherwise, the drone could lead to trajectory deviations or even crash.

B. Definition of Range Specification Bugs

To fulfill missions, users normally refer to the official document (e.g., *Ardupilot*[1] and *PX4*[2]) of instructions provided by manufacturers to alternate the internal configuration parameters. According to the official instructions, each configuration parameter has a range of values demonstrating that the drone can work stably by configuring the parameter within the range. Unfortunately, researchers found that the officially given configuration values could cause unstable flight states [10], [11].

By analyzing the physical impact on drones, we summarized five unstable flight states that configurations might lead to, and the configurations that will cause the following unstable states are defined as *range specification bugs* [10].

Deviation. Ideally, the flight control system will calculate an expected trajectory and predict the next flight states based on the current flight states to guide the drone following the expected trajectory. However, an incorrect configuration may lead the drone to reach a position far from the predicted flight states, triggering a significant position deviation. Such a deviation may lead to an erroneous trajectory from which the drone cannot return to the correct trajectory. Two incorrect scenarios might be triggered: overshooting and flying away. The incorrect configuration may restrict its ability to decelerate, causing an overshoot deviation (see the video in which the drone cannot decelerate at [13]). Besides, when the deviation increases consecutively, the drone will gradually become uncontrollable (i.e., fail to stabilize through manual intervention) and fly away from the planned trajectory (See the video sample of deviating away from the planned trajectory [14]).

After analyzing the manual instruction, we found that when the deviation exceeds 1.5 meters, the possibility that the flight control program failed to drive the drone back to the predicted state will increase. Therefore, we consider a flight state as "deviation" if 15 consecutive deviations exceeding 1.5m are detected.

Flight Freeze. An incorrect configuration might also lead the drone to freeze at a waypoint or wander around a specific position within a minimum movement when a moving forward/backward command is given. The video [15] shows that the drone keeps circling and cannot complete the mission. In addition, its offline flight log demonstrates the drone kept flight switching between *althold* (hovering fly) and *land*. After calculating the movement within a time interval, we regard a flight state as "flight freeze" if the movement is less than 0.5m in 15 seconds.

Drone Crash. Significant deviations will cause the drone to become uncontrollable. Even worse, the drone might eventually be led to an object and crash. The sample video of the drone crash is uploaded [16].

Potential Thrust Loss. Generally, each drone has a specific motor power, and the flight control program adjusts the drone's attitude through the motor power signal. Nonetheless, the actuator is limited to changing the drone angle or speed for flight position recovery. When a drone's attitude achieves a limit that

[1]https://ardupilot.org/copter/docs/parameters.html
[2]https://docs.px4.io/main/en/advanced_config/parameter_reference.html

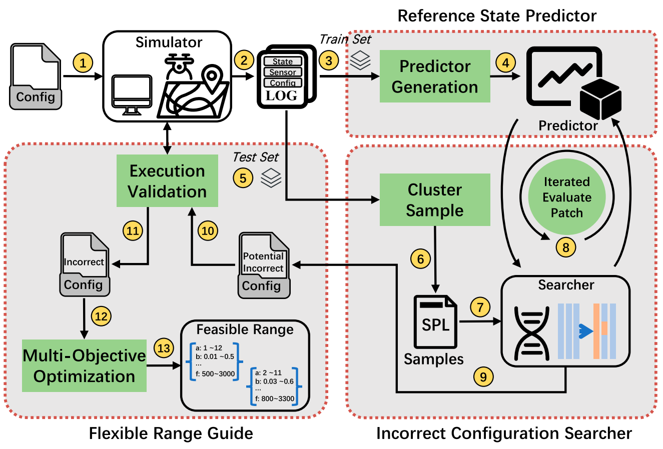


Fig. 2. Overview of ICSEARCHER.

the attitude cannot be driven back to the expected states even if the current throttle has already saturated the motor power up to 100%, This failure may further lead to a drone crash. We consider such a flight state as "thrust loss".

Post-Launch Privilege Escalation. Before taking off, the user needs to set an initial configuration and the flight mission and then send these to the flight control program. The flight control program further validates the parameters related to position control (e.g., `ATC_*_*_P/I/D`, `MC_*RATE_P/I/D`) in the configuration. If a potential instability is predicted, the control program will abort the mission and raise a warning of "motor unlock deny". Unfortunately, such incorrect configurations are acceptable after taking off and then trigger accidents, such as crashes. The demo video is demonstrated [17]. We regard this scenario as "post-launch privilege escalation".

III. ICSEARCHER

To exploit range specification bugs effectively and efficiently, we propose a lightweight detection tool, ICSEARCHER, which leverages genetic algorithm (GA) to carry out mutation fuzzing search and validate impacts of parameters.

A. Overview of ICSEARCHER

Fig. 2 demonstrates the framework of ICSEARCHER containing three phases:

- *Reference State Predictor:* It adopts historical flight data and creates a predictor to predict state change of flight, which is also utilized to assess incorrect configurations;
- *Incorrect Configuration Searcher:* It applies GA search module to produce the potential incorrect configurations;
- *Flexible Range Guide:* It applies multi-objective optimization to provide secure range guidelines.

The generation of the predictor and exploration process of the searcher necessitate diverse flight data. Our system relies on a simulator that repeatedly executes drone flight missions to generate flight data in varying configurations (① ②). The flight log data is divided into two parts. The reference state predictor employs one portion to train the predictor itself (③ ④), while the other serves as foundational data concerning flight states and sensor data for the searcher (⑤). Subsequently, based on foundation data, ICSEARCHER carries out searches for incorrect configurations. ICSEARCHER first clusters the data and sample representatives from each cluster group (⑥ ⑦). The searcher then employs each sample to search to excavate potential incorrect configurations (⑧). During the search process, our system iteratively utilizes the predictor to evaluate which configuration carries a higher probability of inducing unstable states. This iterative search process terminates upon fulfillment of specific conditions. Any identified potential incorrect configurations from the respective sample are collected to form a set (⑨) and subsequently validated through execution in the simulator (⑩ ⑪). Finally, referring to validation results, the flexible range guide leverages a multi-objective optimization approach to refine multiple feasible range guidelines that strike the best balance between availability and stability under specific conditions (⑫ ⑬).

B. Flight Log Generation

Initially, the reference state predictor is a black box model with unknown parameters, requiring numerous data for fitting. Since there is no standard dataset describing the input/output of drones, we establish a flight test scenario to gather flight logs to construct a predictor and provide state samples for further searching. As deviations from default configuration parameters during flights can result in drone crashes, we employed flight simulations to gather our flight logs. Moreover, crashes may still occur even with the default configuration if environmental factors, such as strong headwinds or tailwinds, are not considered. The flight test set the same flight mission, *AVC2013* [18] often used to test the drone mission execution capabilities and uploads with different configurations. We recorded all flight logs but excluded those that lead to unstable states because the flight unstable states caused by the configurations are uncontrollable. The generation of predictor is to imitate the original control logic, and unstable states mean the control logic is out of control; therefore, the flight logs caused unstable states resulting in negative learning experiences for predictor generation. Submodels in drones have different typical log recorded rates. Since the values of the state constantly change, if the sampling rate is too low, a large amount of similar data will be generated; if the sampling rate is too high, value changes in data will not be smooth enough. Additionally, the experimental control hardware writes memory data into the nano flash at 10Hz. Therefore, we use a unified sampling rate 10Hz, equivalent to a 0.1 second interval.

Each log entry contains state information, including angular position, angular acceleration, throttle speed, sensor data of GPS, gyroscopes, accelerometers, current set configuration, and timestamp index. For our discussed flight control program, *Ardupilot* and *PX4*, in total, we recorded $740,799$ system log entries for *Ardupilot* and $410,962$ system log entries for *PX4*.

C. Reference State Predictor

Although leveraging GA fuzzing can reduce parameter search space, the number of parameter combinations (configurations) remains numerous, making validation through realistic/simulation execution highly inefficient. Instead of execution validation, we leverage a predictor to validate the impact of a configuration. The flight control algorithm estimates the next reference state based on previous state feedback, current state, sensor data, and loaded configuration. We apply a machine learning (ML) predictor to emulate that process, referring to their input/output correlation. Unlike the original control algorithm, as the predictor is used to fit the control algorithm to improve accuracy, our predictor considers multiple historical feedback state data as input. Thus, the predictor has the same property control algorithm to estimate the reference state and further assess how the configurations affect flight states. Two steps are executed to train the predictor: *feature extraction* and *predictor generation*.

1) Feature Extraction: Given the log entries, ICSEARCHER constructs a feature matrix by selecting the specific data because not all items in an entry have a variable deviation under the influence of unstable scenarios. For instance, even if the drone's attitude is disturbed, position states (altitude, latitude, longitude) will not change violently. Similarly, temperature sensor data remains essentially constant throughout a mission. Parameters such as `BATT2_VLT_OFFSET` (voltage offset adjusting) and `COMPASS_OFS_X` (compass offsets in milligauss on the X axis) do not affect the attitude. Since we aim to analyze the unstable states that affect flight attitude (i.e., angular and speed), the feature in ICSEARCHER considers the following state, sensor data, and configurations: (i) angular attitude and speed of the flight state a; (ii) sensor data s obtained from gyroscopes, accelerometers and compasses; (iii) control or mission parameters x that regulate attitude. According to the above information, ICSEARCHER groups them as a feature vector, $v\{a, s, x\}$. In further searching, x and a change together. For convince of description, we manually define $c = \{a, s\}$. The feature vector in timestamp t is $v(t) = \{c(t), x(t)\}$. And the values are normalized to [0, 1] with a min-max scaler. The vectors are further combined to construct a feature matrix.

2) Predictor Generation: As Long Short-Term Memory (LSTM) [19] technique handles complicated input/output data efficiently [20], [21], [22], ICSEARCHER further utilizes *LSTM* as the predictor to refer to the next reference state. Specifically, the predictor takes a number h of consecutive vectors whose timestamp is lower or equal than t as input (i.e., $V\{v(t-h-1), ..., v(t-1), v(t)\}$), and estimates the next reference state units $a'(t+1)$ in $v'(t+1)$. In Section IV, we evaluate the effect of the vector size h on the prediction quality. In the training stage, we employ *Mean Squared Error (MSE)* [23] to iteratively optimize the internal weights in the predictor, ensuring the predicted state $a'(t+1)$ is closer to the ground truth state $a(t+1)$ enough.

Note that though the predictor is specified to a certain control program, the prediction scheme is extensible based on the corresponding control programs (i.e., dynamic models, control algorithms, configuration parameters, and data sampling methods). The predictor generated for a certain control program cannot be used directly for others. However, the template for generating predictors is generic; thus, the different flight control programs can extract the same format state and sensor data to instantiate the predictor for estimating the reference state.

D. Incorrect Configuration Searcher

To explore incorrect configurations that trigger unstable deviation, i.e., a severe deviation between the reference state and the current state, we leverage historical flight data and launch searches to discover potential incorrect configurations with a high probability of causing state deviation. We adopt Genetic Algorithm (GA) [12], a metaheuristic search relying on biologically inspired operators such as mutation, crossover, and selection. Since configurations may only produce unstable states in specific contexts (i.e., state and sensor data), ICSEARCHER conducts searches for different contexts to find corresponding incorrect configurations.

Initially, for raw n flight entries $Raw = \{c(1), c(2), ...c(n)\}$, we split them into multiple segments $C(i)$ as contexts. Each segment is generated with the following:

$$C(i) = \{c(i), c(i+1), ..., c(i+len-1)\} \tag{1}$$

where i is the index of the recorded data, and len is the length of each segment. Since our predictor accepts multiple parameter values as input with the length of h, the length of the segment should not be too small. Using multiple data to form a segment can eliminate the bias of (using) single data (transient) prediction errors. However, a large segment length will affect the efficiency of the fuzzing search process, indicating more data in the segment is required to be predicted. Considering that our data sampling frequency is 0.1 second (10Hz), every 10 entries can represent the state change of the drone in one second, and through our experimental observation, we found that one second of data is enough to show the trend of drone state change. Therefore, to balance prediction performance in the fuzzing process and reduce the negative effects of prediction, we set the length of each segment to $10 + h$ (i.e., $len = 10 + h$), which will get 10 deviation values.

Since similar segment data with duplicated configurations may reduce search efficiency, ICSEARCHER adopts the method of clustering first and then sampling for selecting representative samples to search, which reduces redundancy while maintaining diversity. ICSEARCHER applies *DBSCAN* [24], a probability density-based non-parametric adaptive clustering algorithm, to cluster segments and randomly sample m representatives from each cluster group. For each specific segment, ICSEARCHER launches GA searcher to predict corresponding incorrect configurations based on mutation, crossover, and selection. In the final, each search result is merged and deduplicated as a unique set referred to as *set of potentially incorrect configurations*. In what follows, we describe details on key ideas, *predictor segment evaluation* about how to assess a configuration, and *searching process* about how the searcher works.

1) Predictor Segment Evaluation: The GA search applies a fitness function to quantify, by using the predictor, how much deviation a given configuration may cause in a specific context (state and sensor data). The predictor segment evaluation creates a score about the predicted deviation this configuration may cause. A higher score indicates that the predictor considers this configuration to be more "dangerous", otherwise, it is more "safe".

Specifically, assume that the current search is carried out for the context segment $C\{c(1), c(2)...c(10+h)\}$. 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 $V\{\{c(1), x\}, ..., \{c(10+h), x\}\}$. Then, the function divides V into patches of length $h+1$ with a sliding window:

$$\{p(1), p(2), ..., p(10)\} \tag{2}$$

where $p(i)$ is:

$$p(i) = \{v(i), v(i+1), ..., v(i+h)\}, i \in [1, 10] \tag{3}$$

For each patch $p(i)$, $p_{in} = \{v(i), v(i+1), ..., v(i+h-1)\}$ is used as input, and $a(i+h)$ in $v(i+h)$ is used as ground-truth. That is, features p_{in} are given as input to the predictor, which estimates a reference state $a'(i+h)$. One deviation is the L1-distance $d = \|a(i+h) - a'(i+h)\|$ between the predicted state and the ground truth state. It creates 10 inputs and ground truth outputs, and a segment has 10 deviations $D = \{d(1), d(2), ..., d(10)\}$, i.e., deviations within 1 seconds. The fitness score of this configuration x is the summation of these deviations D_{sum}. The target of the search is to search configurations predicted to maximize the fitness, that is, maximize the probability of causing unstable states.

Using segment deviation instead of single deviation considers the predictor's accuracy. Although the predictor can maintain a high prediction accuracy rate, it is still possible to have a prediction bias because of some extreme data, thus affecting the judgment of the configuration evaluation. However, segment deviation will reduce the weight of a single deviation on the overall evaluation. Even if one prediction is wrong, it will not affect the overall evaluation metrics. Therefore, for the fuzzing process, such an evaluation function may promote the accuracy of the searcher to find actual incorrect configurations.

2) Searching Process: For current segments $C(i)$, initially, the searcher first assigns the population (configuration group) with default parameter values, where individuals in the population represent configurations. Assuming the population size is NP, and the maximum number of iterations is G_{max}. The search process iteratively mutates and updates this population as follows.

Assuming the current iteration is g-th ($g \in [1, G_{max}]$), 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:

$$y_{i,g} = x_{i,g} + F * (x_{best,g} - x_{i,g}) + F * (x_{r1,g} - x_{r2,g}) \tag{4}$$

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

Searcher then carries out cross operation for pop_g and pop_v to produce a new experimental population $pop_e = \{e_{1,g}, ..., e_{NP,g}\}$, where $e_{i,g} = \{e_{i1,g}, e_{i2,g}, ..., e_{ij,g}\}$, $i \in [1, NP]$, $j \in [1, D]$. The parameter values $e_{ij,g}$ in a configuration are obtained as follows:

$$e_{ij,g} = \begin{cases} y_{ij,g}, & if\ rand(0,1) < CR\ or\ j = j_{rand} \\ x_{ij,g}, & otherwise \end{cases} \tag{5}$$

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 applies *predictor segment evaluation* to calculate the score of configurations both in pop_g and pop_e. According to their score, the searcher selects configurations from the population to create the next generation pop_{g+1} with the following method:

$$x_{i,g+1} = \begin{cases} e_{i,g}, & if\ f(e_{i,g}) < f(x_{i,g}) \\ x_{i,g}, & otherwise \end{cases} \tag{6}$$

where f is *predictor segment evaluation*.

Finally, if the fitness of the population no longer increases, or iteration reaches the maximum number G_{max}, the searcher stops, and the top 10 configurations with the highest score from the final generation population are perceived as potentially incorrect configurations.

E. Flexible Range Guide

Depending on the potential incorrect configurations provided by the searcher, ICSEARCHER needs to summarize relatively safe new range guidelines that minimize the probability of triggering unstable flight states.

1) Execution Validation: The search results contain potentially incorrect configurations that have a high probability of causing unstable states. Whether they are incorrect requires to be validated through flight execution. Thus, we leverage a flight simulator platform to execute these configurations and observe which lead to unstable states. Similar to the collection phase, the drone set with these configurations to carry out *AVC2013* mission.

2) Range Guide Generation: The target of the parameter range is to prevent the user from uploading unstable configurations. Besides, it should also consider stability and adaptability. Users may have preferred values for certain parameter values, and the range provided should consider a relatively safe range while satisfying the user's requirement as much as possible. Since we cannot ensure the stability of unverified configurations, the guide generation should reference the execution validation results while considering their incorrect configurations. Therefore, rather than providing a specific range guideline, we leverage multivariate optimization to determine flexible range guidelines. If all parameter values in a configuration are within the range specified by the guideline, we consider the configuration to be covered. ICSEARCHER perceives the generation of range guidelines ($Range'$) as the following optimization problem:

$$\begin{cases} min\ f_1 = \frac{num(R_{incorrect} \in Range')}{num(R \in Range')} \\ max\ f_2 = num(R \in Range') \\ s.t. \\ Range' \in OriginRange \end{cases} \tag{7}$$

The optimization problem involves two objectives: (a) Utilize validation results as possible to maximize the number of validated configurations R covered by the guideline; (b) Keep as safe as possible to minimize the coverage of incorrect configurations $R_{incorrect}$. Shrinking the coverage of incorrect configurations would also shrink the range of each parameter value and vice versa. Therefore, instead of defining strict ranges for each parameter, the system solves this optimization problem with multiple constraints to construct a diverse group of *Pareto* boundary solutions. These *Pareto* solutions form a boundary consisting of the best feasible solutions, allowing users to choose the best configuration to balance their requirements. ICSEARCHER provides flexible range guidelines 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. Therefore, users can first select the guide that satisfies their specific value requirements from the flexible range and choose the one with the higher security as a reference for setting other parameter values.

IV. Evaluation

We assessed ICSEARCHER by answering the following research questions (RQs):

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

A. Experiment Setup

We applied ICSEARCHER to two prevalent open-source flight control programs, *ArduPilot* (4.2.0) [5] and *PX4* (1.13) [3], which are widely used by drone manufacturers such as *Parrot*, *Mamba*, and *Mateksys* [25]. To validate configurations specified in *ArduPilot* and *PX4*, we utilized four experimental vehicles (shown in Fig. 3) for testing, including two drones with *Pixhawk* [26] (i.e., *CUAV ZD550* and *AMOVLab Z410*), two drone simulator (i.e., *Airsim* [27], *Jmavsim* [28]).

According to the control parameter descriptions provided by the manufacturer, we selected 20 parameters for *Ardupilot* (Table I) and 14 parameters for *PX4* (Table II) that may affect angular flight position and angular speed. Note that a predictor is valid for a specific flight control program, so for *Ardupilot*

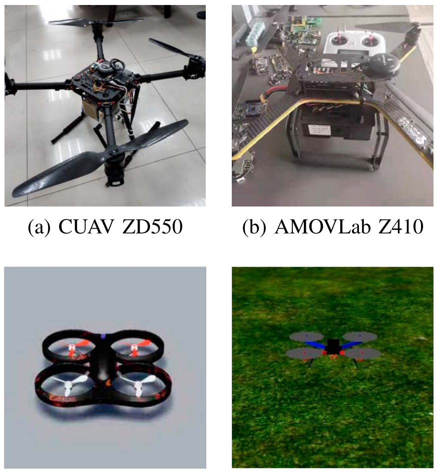

(a) CUAV ZD550 (b) AMOVLab Z410 (c) Airsim (d) Jmavsim

Fig. 3. Real and virtual drone vehicles used for experiments.

TABLE I
PARAMETERS OF ARDUPILOT CONTROL PROGRAM FOR EXPERIMENTS

Parameter	Range	Default
PSC_VELXY_P	[0.10, 6.00]	2.0
PSC_VELXY_I	[0.02, 1.00]	1.0
PSC_VELXY_D	[0.00, 1.00]	0.5
PSC_ACCZ_P	[0.20, 1.50]	0.5
PSC_ACCZ_I	[0.00, 3.00]	1.0
ATC_ANG_RLL_P	[3.00, 12.0]	4.5
ATC_RAT_RLL_P	[0.01, 0.50]	0.135
ATC_RAT_PIT_I	[0.01, 2.00]	0.135
ATC_RAT_RLL_D	[0.00, 0.05]	0.0036
ATC_ANG_PIT_P	[0.00, 12.0]	4.5
ATC_RAT_PIT_P	[0.01, 0.50]	0.135
ATC_RAT_PIT_I	[0.01, 2.0]	0.135
ATC_RAT_PIT_D	[0.00, 0.05]	0.0036
ATC_ANG_YAW_P	[3.00, 12.0]	4.5
ATC_RAT_YAW_P	[0.10, 2.50]	0.18
ATC_RAT_YAW_I	[0.01, 1.00]	0.018
ATC_RAT_YAW_D	[0.00, 0.02]	0
WPNAV_SPEED	[20, 2000]	500
WPNAV_ACCEL	[50, 500]	100
ANGLE_MAX	[1000, 8000]	4500

and *PX4*, we used the same architecture combined with their respective data to produce predictors separately.

The predictor and the GA searcher are implemented in Python. Concerning the GA, its evolutionary stagnation judgment threshold is set to 0.1, the number of representatives m is set to 10, and the maximum number of evolutionary generations is set to 200. We further developed the initial population size to 500 (i.e., $NP = 500$) and the scaling factor F to 0.4. Our LSTM model consists of an LSTM Cell, a dropout layer (on 0.1 drop rate), a Dense layer with ReLU activation, and one fully connected output layer.

B. RQ1: Effectiveness

To evaluate whether ICSEARCHER identifies incorrect configurations accurately, we first conducted the assessment on both *Ardupilot* and *PX4* to calculate the prediction accuracy

TABLE II
PARAMETERS OF PX4 CONTROL PROGRAM FOR EXPERIMENTS

Parameter	Range	Default
MC_ROLL_P	[0.00, 12.0]	6.5
MC_PITCH_P	[0.00, 12.0]	6.5
MC_YAW_P	[0.00, 5.0]	2.8
MC_YAW_WEIGHT	[0.00, 1.00]	0.4
MPC_XY_P	[0.00, 2.00]	0.9
MPC_Z_P	[0.00, 1.50]	1.0
MC_PITCHRATE_P	[0.01 0.60]	0.15
MC_ROLLRATE_P	[0.01, 0.50]	0.15
MC_YAWRATE_P	[0.00, 0.60]	0.2
MPC_TILTMAX_AIR	[20.0, 89.0]	45.0
MIS_YAW_ERR	[0.00, 90.0]	12.0
MPC_Z_VEL_MAX_DN	[0.5, 4.0]	1.0
MPC_Z_VEL_MAX_UP	[0.5, 8.0]	3.0
MPC_TKO_SPEED	[1.0, 5.0]	1.5

TABLE III
PREDICTOR ACCURACY IN TEST DATA WITH DIFFERENT INPUT LENGTHS

h	2	3	4	5	6
Ardu	96.05%	96.19%	97.10%	95.60%	95.77%
PX4	94.32%	95.56%	96.15%	94.52%	94.45%

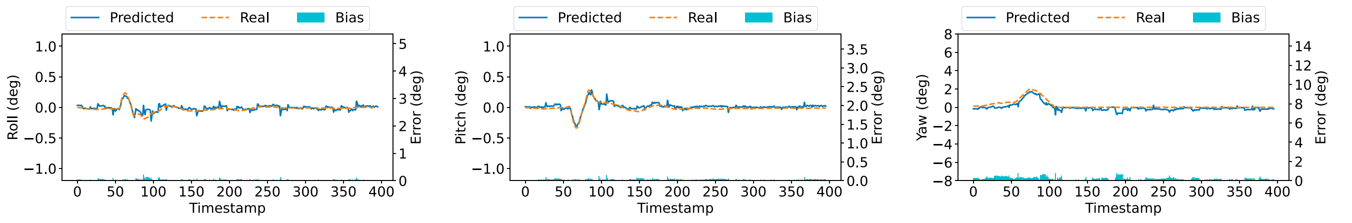
*Ardu is Ardupilot.

of the predictor in the flight state predictor phase. Then, we validated whether the predicted incorrect configurations could impact flight stability.

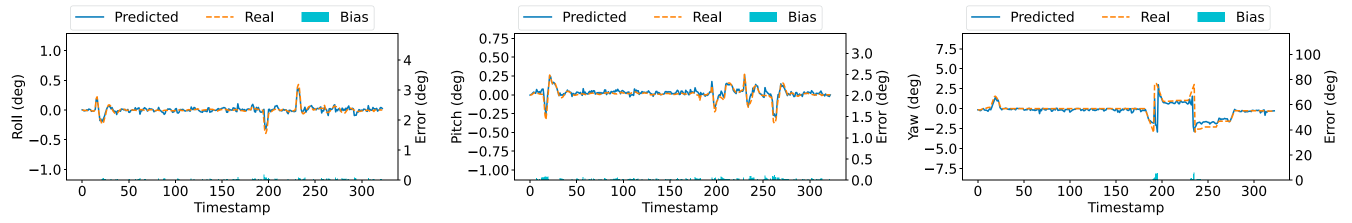
1) State Prediction: We began by utilizing the log data from Section III-B to create a predictor for both *Ardupilot* and *PX4*, with 90% of the data used for training and 10% for testing. We then determined the optimal input length h. The test results are shown in Table III. Based on these results, we selected the predictor with the best accuracy, the input length of h=4, to conduct subsequent experiments.

In addition, we demonstrate the difference between the actual and predicted state-of-charge using two *real-world* flight logs, one for *Ardupilot* and another for *PX4*. We present the consecutive values of both the predicted and actual state in Fig. 4. In the figure, the dotted curve denotes the actual states, the solid curve denotes the predicted states, and the histogram at the bottom (the cyan bar) indicates the prediction errors as differences between the two curves. The figure demonstrates that the predictor accurately predicts the trend for state changes, closely matching the actual values.

As outlined in our methodology, a suitable predictor should be able to distinguish between stable and unstable states accurately. This property enables the predictor to facilitate fuzzing by identifying configurations with a higher probability of triggering unstable states. To demonstrate the effectiveness of our predictor, we conducted a simple threshold classification experiment using Ardupilot as an example. We then compared our predictor's performance to that of LGDFUZZER's prediction function. LGDFUZZER estimates the likelihood of a configuration leading to unstable states based on a single deviation between the predicted and actual values. On the contrary, ICSEARCHER



(a) Real-World Ardupilot State (Roll, Pitch, and Yaw)



(b) Real-World PX4 State (Roll, Pitch, and Yaw)

Fig. 4. Prediction match of Ardupilot and PX4.

TABLE IV
STABLE/UNSTABLE AVERAGE DEVIATION AND CLASSIFICATION PERFORMANCE

	ST	UST	TH	Precision	Recall	F1
Seg	1.72	4.77	3.25	92.48%	98.19%	95.25%
Sin	0.15	0.34	0.24	80.99%	98.19%	89.01%

***Seg** is segment method in ICSEARCHER. **Sin** is single method in LGDFUZZER. **ST** is the average value from stable data ST_{avg}. **UST** is the value from unstable data UST_{avg}. **F1** is F1-score.

TABLE V
DETAILS OF TEST DATA

Program	Log Entry	Segment	Cluster	Selected Samples
Ardupilot	74,074	5,698	58	571
PX4	41,096	3,161	33	316

calculates the segment deviation (sum of deviations within the segment) to make this determination.

Specifically, we conducted 2,000 configuration tests, of which 1,542 did not complete the mission due to critical incidents, while 458 completed their mission successfully. The experiment used 1,600 out of the 2,000 configurations (considering category weights) to calculate the threshold, with the remaining 400 configurations used for testing purposes. When analyzing test logs that contain critical incidents (which we mark as unstable), ICSEARCHER extracts the deviation segment that occurred before the crucial incident timestamp. In contrast, LGDFUZZER only considers a single deviation metric that occurred before the crucial incident timestamp. For stable test logs that do not contain critical incidents (we mark them as stable), ICSEARCHER randomly recorded a deviation segment, while LGDFUZZER selected a single deviation metric at random. To determine the threshold, we calculated the average segment/single deviation for both stable (ST_{avg}) and unstable (UST_{avg}) data. We then set the threshold to be the midpoint between these two average values, i.e., $(ST_{avg} + UST_{avg})/2$.

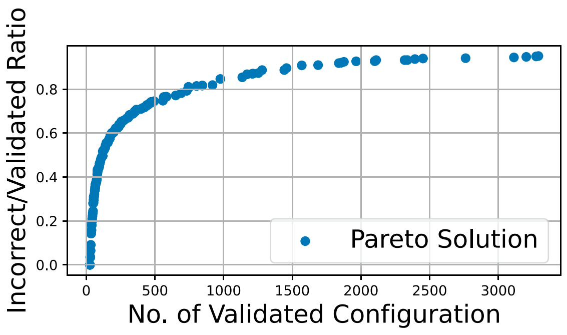
Table IV shows the calculated threshold through 1,600 data and unstable detection performance in 400 test data. **Seg** is used to indicate a segment deviation value (ICSEARCHER), and **Sin** indicates a single deviation value (LGDFUZZER). Based on their respective average deviation values exhibited in the table, the threshold for the segment method is 3.25, while the threshold for the single method is 0.24. The test data exceeds the threshold and is marked as unstable.

The deviation segment and single deviation analyses show that unstable states exhibit greater deviations than stable states. This phenomenon supports using the predictor in GA searches, as it can drive configurations towards higher deviation values and more effectively identify unstable states. Compared to the previous LGDFUZZER method, which failed to achieve an F1-score exceeding 90%, ICSEARCHER performs better in terms of F1-score. This improvement can be attributed to the use of segment deviation. By aggregating deviations over a segment, the impact of predictor accuracy is mitigated, and the numerical value of the deviation fluctuates less. In other words, using the segment method in fuzzing can more accurately predict whether a configuration will cause an unstable state.

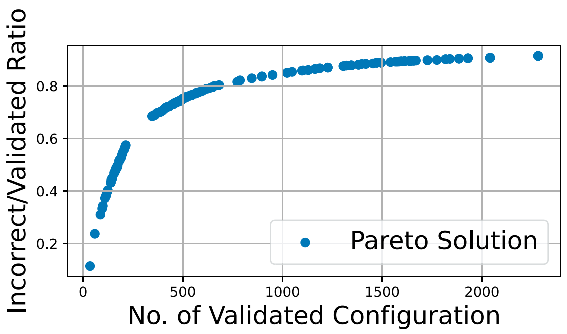
2) Configuration Validation: We applied test data, comprising 10% of the log data as specified in Section III-B (Table V), to conduct configuration validation experiments. The dataset included 74,074 log entries (5,698 segments) for *Ardupilot* and 41,096 log entries (3,161 segments) for *PX4*. These segments are clustered into 58 and 33 clusters. Using ten samples extracted from each cluster (or all available samples if less than 10), a total of 571 samples for further testing were generated for *Ardupilot* and 316 for *PX4*.

Then, we launched fuzzing searches for these segments to find potential incorrect configurations. Finally, we validated them by sending those configurations to the flight control programs. For *Ardupilot*, ICSEARCHER explored $4,386$ unique incorrect configurations. After validation, 4,157 out of 4,386 configurations were marked as truly incorrect resulting including 2,150 *Deviations*, 432 *Flight Freeze*, 9 *Crashes*, 2 *Post-Launch Privilege Escalation*, and 1,564 *Potential Thrust Losses*, respectively. Similarly, for *PX4*, ICSEARCHER found out 2,282 unique incorrect configurations. After validation, 2,087 out of 2,282 configurations were marked as truly incorrect configurations including 641 *Deviations*, 228*Flight Freeze*, 309 *Crashes*, and 909 *Post-Launch Privilege Escalation*, respectively.

The results suggest that *Ardupilot* is more prone to causing trajectory deviation and potential thrust losses, whereas *PX4* exhibits a higher rate of post-launch privilege escalation and crashes. There are differences between the experimental results of these two due to flight control program design, like raw data acquired from sensor drivers, parameter lists, flight control algorithms, and log structure, so the configurations do not always show similar results. For instance, by design, the *PX4* additionally employs a model to check whether the current configuration will result in a theoretical maximum speed that exceeds the threshold. And if so, the control program will prohibit the drone from taking off. However, these configurations will still be accepted if sent during flight, resulting in more post-launch privilege escalation.



(a) Pareto frontier solution of Ardupilot experiment



(b) Pareto frontier solution of PX4 experiment

Fig. 5. Pareto frontier solution.

C. RQ2: Adaptability

To demonstrate the adaptability of the flexible range guide of ICSEARCHER, we leverage the results of confirmed incorrect configurations (i.e., 4,157 *Ardupilot* and 2,087 *PX4*) to generate range guidelines. Referring to the validation results, the flexible range guide provides 178 *Pareto* for *Ardupilot* and 213 for *PX4* solutions. Fig. 5 demonstrates the validation result. The horizontal axis represents the number of validated configurations covered by the flexible range guideline, and the vertical axis represents the ratio of incorrect configurations in the flexible range guideline. Each *Pareto* solution represents a suggested configuration range in the figure. The further the solution is to the left, the smaller the probability that the range contains incorrect, and the smaller the available values space. Conversely, the solution is to the right, and the probability of incorrect configurations in the range is greater while providing the wider available values space.

For instance, we chose some representative examples (*Pareto* solution) to demonstrate the stability and adaptability of different guidance. By selecting some representative guideline parameters as examples, we demonstrated the feasible value ranges of *Ardupilot* and *PX4* in Table VI, where reduce means how much the new guideline has been reduced in range compared to the official one (i.e., adaptability), and the more it is reduced, the less adaptable it is; $\frac{I}{V}(\frac{Incorrect}{Validated})$ indicates the percentage of incorrect configuration verified in the guideline (i.e., stability); lower means more secure.

The table respectively shows three examples in which, from i to iii, their stability decreases and the configurable space (i.e., adaptability) increases. For *Ardupilot* configurations, *Guideline* i avoids the majority of incorrect configurations, which covers 26 validated configurations. Among the configurations, none caused unstable states, which indicates high stability. However, stricter range guidance will reduce system adaptability compared with the original parameter ranges. In contrast, *Guideline* iii achieves higher adaptability by covering 52 valid configurations; however, it covers 26.9% incorrect configuration, which resulted in low stability. *Guideline* ii is an intermediate choice, covering 29 validated configurations where only 3.4% are incorrect. Similarly, for *PX4* configurations, *Guideline* i has the highest security but the smallest available range, which covers 26 validated configurations, and only 4.0% is incorrect. *Guideline* ii and *Guideline* iii separately cover 32 and 93 validated configuration, and have 9.3% and 32.2% incorrect configuration.

Stability and adaptability requirements, users are free to choose an appropriate range of guidelines from the *Pareto* solution according to their stability and adaptability requirements; that is if the user has more stringent stability requirements, a lower error rate range guideline can be used at the cost of limited configuration space for other flight requirements. Conversely, if the user has special mission requirements (e.g., the mission is time-limited or requires a large flight angle to reach the target speed), in that case, they may consider sacrificing stability for improved adaptability. For example, if the user requires the parameter `MPC_Z_P` to be 0.2 in their mission requirements, he/she can select *Guideline* ii in PX4 instead of *Guideline* i. Although *Guideline* ii is not the safest option, its range for `MPC_Z_P` (0.0, 1.4) prioritizes meeting the required value. The user can use this guideline to determine other parameters as well.

TABLE VI
EXAMPLES OF FEASIBLE RANGE GUIDELINE

Control Program	Parameter	*Guideline* i			*Guideline* ii			*Guideline* iii		
		Low	**Up**	Reduce	**Low**	**Up**	Reduce	**Low**	**Up**	Reduce
Ardupilot	PSC_VELXY_P	0.3	5.9	-5.1%	0.3	6.0	-3.3%	0.3	6.0	-3.3%
	PSC_ACCZ_I	0.0	2.9	-3.3%	0.0	2.4	-20.0%	0.0	2.6	-13.3%
	ATC_RAT_RLL_P	0.13	0.37	-51.0%	0.125	0.375	-48.9%	0.12	0.38	-46.9%
	ATC_RAT_RLL_I	0.01	0.46	-77.3%	0.015	0.445	-78.3.%	0.01	0.915	-54.5%
	ATC_RAT_PIT_P	0.05	0.50	-8.1%	0.05	0.50	-8.1%	0.055	0.475	-14.2%
	ATC_ANG_YAW_P	3.1	11.9	-2.2%	3.1	12.0	-1.1%	3.0	12.0	-0.0%
	WPNAV_SPEED	750	2000	-35.8%	700	2000	-33.3%	800	2000	-38.4%
	ANGLE_MAX	1000	7020	-14.0%	1000	6630	-19.5%	1000	6920	-15.4%
	$\frac{I}{V}$,**V**	0.0%, 26			3.4%, 29			26.9%, 52		
PX4	MC_ROLL_P	1.0	12	-8.3%	1.4	11.9	-14.1%	1.2	12.0	-0.1%
	MC_PITCH_P	0.5	12	-4.1%	3.1	12.0	-4.1%	1.8	11.9	-15.8%
	MC_YAW_P	1.0	5.0	-20.0%	0.7	4.8	-2.0%	0.0	4.9	-2.0%
	MPC_Z_P	0.5	1.4	-40.0%	0.0	1.4	-6.7%	0.2	1.4	-20.0%
	MC_ROLLRATE_P	0.13	0.49	-26.5%	0.08	0.50	-14.2%	0.06	0.5	-10.2%
	MPC_Z_VEL_MAX_DN	0.9	3.3	-31.4%	0.9	3.6	-2.8%	0.6	3.9	-5.7%
	$\frac{I}{V}$,**V**	4.0%, 26			9.3%, 32			32.2%, 93		

***Low** is range lower bound, **UP** is range upper bound, reduced is calculated relative to original range (smaller means higher adaptability), **I** is the number of incorrect configurations covered by the range guidance, **V** is the number of validated configurations covered by the range guidance, $\frac{I}{V}$ is the ratio of incorrect configurations in the range guidance (smaller means higher stability).

TABLE VII
PARAMETERS OF ARDUPILOT CONTROL PROGRAM FOR COMPARISONS

Parameter	Range	Default
PSC_VELXY_P	[0.0, 6.0]	2.0
INS_POS1_Z	[-5.0, 5.0]	0.0
INS_POS2_Z	[-5.0, 5.0]	0.0
INS_POS3_Z	[-5.0, 5.0]	0.0
WPNAV_SPEED	[20, 2000]	500
ANGLE_MAX	[1000, 8000]	4500

D. RQ3: Improvement

To demonstrate the advantages of ICSEARCHER, we experimentally compared it with the state-of-the-art tool, RVFUZZER [10], which leverages two methods *One-dimensional Mutation* and *Multi-dimensional Mutation* to search incorrect configurations. Besides, we also compared range guidelines generated with LGDFUZZER [11]. All these experiments were based on the six parameters utilized in RVFUZZER (see Table VII). In the experiments, we considered the unstable states listed in Section II-A.

1) Comparison With Respect to Search Missed: The optimal solution of RVFUZZER may not be consistent with the right optimum. For instance, *One-dimensional Mutation*, indicated the correct range for `INS_POS1_Z` should remain within $[-4.7, 0.0]$ compared to original lower bound $[-5.0, 0.0]$. However, in our validations, there were still incorrect configurations inside this range, especially between -1.0 and 0.0 In our analysis, the binary search directly skipped values greater than -2.5 since the first midpoint (i.e., -2.5) did not cause unstable states, which can not cover the $[-1.0, 0.0]$ space, resulting in missing incorrect configurations. In the case of multiple parameter mutations, we first determine a prior correct range through *Multi-dimensional Mutation*. Based on this range we then leveraged ICSEARCHER to start another search to excavate incorrect configurations and still detected 106 incorrect configurations. Such a result indicated that the range provided by *Multi-dimensional Mutation* had certain flaws. In our opinion, the *Multi-dimensional Mutation* is still a one-dimensional 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.

2) Comparison With Respect to Range Guidelines: We leveraged ICSEARCHER to identify incorrect configurations for these six parameters. The results reported 1,077 unique potentially incorrect configurations, and after validations, 806 out of them are determined to lead to unstable states. After that, we used them to generate the feasible range guidelines and chose the safest one for comparison. Table VIII lists the ranges obtained by four methods, containing *1* (*One-dimensional mutation*), *M* (*Multi-dimensional mutation*), ICSEARCHER, and LGDFUZZER.

RVFUZZER method *1* obtained little reduction for each parameter range; thus, it can not rule out incorrect configurations. *M* avoided some incorrect configurations but still missed others. LGDFUZZER has higher safety, but since its search for the potential incorrect configuration is not as accurate as ICSEARCHER, the most secure range generated with the same data is not as good as ICSEARCHER. Our approach ICSEARCHER provided high stability that eliminated most validated incorrect configurations. A special case is related to `INS_POS2_Z`, the lower bound range given by ICSEARCHER is smaller than the one given by

TABLE VIII
COMPARISON OF RANGE GUIDELINE

Parameter	*1*		*M*		LGDFUZZER		ICSEARCHER	
	Low	**Up**	**Low**	**Up**	**Low**	**Up**	**Low**	**Up**
PSC_VELXY_P	0.1	6.0	0.6	6.0	1.9	4.2	0.7	4.1
INS_POS1_Z	-4.7	5.0	-1	4.1	-0.5	2.1	-0.7	0.8
INS_POS2_Z	-5.0	5.0	-0.7	3.2	-0.7	1.2	-0.8	0.7
INS_POS3_Z	-5.0	5.0	-0.8	3.0	-0.9	3.2	-0.3	0.4
WPNAV_SPEED	50	2000	300	2000	50	1950	300	1950
ANGLE_MAX	1000	8000	1000	8000	1100	4650	1000	6850
$\frac{I}{V}$,**V**	74.8%, 1077		51.4%, 206		28.5%, 14		0.0%, 9	

1*: One-dimensional mutation, *M*: Multi-dimensional mutation, **Low is range lower bound, **UP** is range upper bound, **V** is the number of validated configurations covered by the range guidance, $\frac{I}{V}$ is the ratio of incorrect configurations in the range guidance (smaller means higher stability).

M. The smaller ranges of other parameters provide more range space for `INS_POS2_Z` since the validity of configurations is decided based on multiple parameters instead of a single one.

`ANGLE_MAX` influences the flight inclination angle, and a too-large value is not for posture adjustment for position adjustment. The too-small value of `WPNAV_SPEED` makes the drone more likely to freeze; both *M* and *GA* reduce this parameter's lower part of the range. The ranges for `INS_POS*_Z` returned by ICSEARCHER are smaller than the ones provided by *M*, which are closer to default values. In fact, changing `INS_POS*_Z` influences the position judgment of the inertial measurement unit. In the real application, these parameters 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.

3) Comparison With Respect to Time Requirements: We analyzed the time taken by ICSEARCHER, LGDFUZZER, and RVFUZZER shown in Table IX. ICSEARCHER and LGDFUZZER applied the same data, which started multiple simulations and took 696 seconds to gather flight logs. In the predictor generation phase, ICSEARCHER took 703 and LGDFUZZER took 700 seconds to train until reaching convergence. And then, the searcher of ICSEARCHER and LGDFUZZER separately took another 244 and 156 seconds for iteration. With six-thread verification (RVFUZZER can not use multiple threads due to its binary search), they took 35 and 33 minutes separately. The difference is that 94.7% of the potential incorrect configurations searched by ICSEARCHER were confirmed as true, compared to 84.7% by LGDFUZZER; in other words, ICSEARCHER is more accurate than LGDFUZZER in the case of approximate time consumption. Ultimately, ICSEARCHER took 50 minutes, and LGDFUZZER took 47 minutes. It is important to note that data labeling and predictor generation are one-time costs.

In comparison, depending on configurations, RVFUZZER takes 20 to 120 seconds to complete a flight test mission. In our experiment, RVFUZZER took 126 minutes to generate guidelines for the six parameters. Additionally, when the number of parameters increases, there is a discrepancy in the resource consumption between RVFUZZER and ICSEARCHER. Specifically, for each additional parameter, ICSEARCHER experiences a linear increase of approximately 10 to 20 seconds in the search phase. In contrast, RVFUZZER takes exponentially longer, roughly doubling the time.

TABLE IX
TIME CONSUMPTION OF DIFFERENT TOOLS

Tool	Pre-Train	Search	Verify (6Thread)	Total
RVFuzzer	-	126min	-	126min
LGDFuzzer	700s	156s	33min	47min
ICSEARCHER	703s	241s	35min	50min

E. Data Availability

The source code of ICSEARCHER is available in web https://figshare.com/articles/software/Source_code_of_ICSearcher_/24947442. The web page briefly describes the experimental environment and requirements of this tool. It also provides an introduction to the functionality of the respective files.

V. RELATED WORK

A. Drone Fault Detection

To prevent errors during flight, previous researchers have proposed numerous methods. Among them, an invariant approach [29] analysis the flight dynamics model and create a control invariant to identify physical attacks against robotic vehicles. However, their approach does not consider attacks that exploit the range specification bugs. To detect faulty components in a drone system, a fault detection method [30] constructs an observer model to estimate the output in fault-free conditions from the historical inputs and outputs. It adopted the deviation between output and predicted values to identify faulty sensor and actuator components. But it still can not process the unstable states raised by range specification bugs. A neural network-based validation approach [31] leverages the analytical redundancy between flight parameters to detect sensor faults, but it only considers external factors. In comparison, our system implements a search system to avoid configurations that are unstable factors within the system.

B. Learning-Based Testing

From the perspective of learning-based testing, there are four relevant systems or methods. NEUZZ [32] leverages a gradient-guided search strategy to cover more test space. It uses a feed-forward neural network to approximate program branching behaviors, but the predictive model is not used for guiding testing. ExploitMeter [33] uses dynamic fuzzing tests with machine learning-based prediction to update the belief in software exploitability. A learning-guided fuzzing [34] applies LSTM to imitate the input-output relation and then leverages a meta-heuristic method to search specific commands that could drive the system into unsafe states. Q-learning [35] algorithm was also applied in reinforcement fuzzing [36] to generate an optimized policy to test PDF processing programs. DeepSmith [37] uses extensive open-source code to train a generative model and uses the model to generate test inputs to automatically check the OpenCL compiler. These methods leverage prior knowledge to drive mutational inputs. Likewise, we introduce machine learning models to guide the search testing process but combine the model with a GA for large-scale multi-parameter searches.

VI. Conclusion

Incorrectly configured drone control parameters can cause unstable flight states that in the worst case can disrupt drone missions. In this paper, we propose a fuzzing-based system that efficiently and effectively searches for potentially incorrect parameter configurations. ICSEARCHER applies a machine learning predictor to assist the fuzzing search, a genetic search algorithm. Finally, through multi-objective optimization, it provides correct feasible ranges. We have experimentally compared ICSEARCHER with a state-of-the-art tool. The experimental results show our system is superior to such a tool in all respects.

References

[1] I. Bekmezci, O. K. Sahingoz, and Ş. Temel, "Flying ad-hoc networks (FANETs): A survey," *Ad Hoc Netw.*, vol. 11, no. 3, pp. 1254–1270, 2013.

[2] P.-B. Bok and Y. Tuchelmann, "Context-aware QoS control for wireless mesh networks of UAVs," in *Proc. 20th Int. Conf. Comput. Commun. Netw. (ICCCN)*, Piscataway, NJ, USA: IEEE Press, 2011, pp. 1–6.

[3] DroneCode, "Px4—An open source flight control software for drones and other unmanned vehicles," PX4. Accessed: 2023. [Online]. Available: https://px4.io/

[4] LibrePilot, "LibrePilot: A software suite to control multicopter and other RC-models," LibrePilot. Accessed: 2023. [Online]. Available: https://www.librepilot.org/site/index.html

[5] ArduPilot Development Team, "ArduPilot—Versatile, trusted, open autopilot software for drones and other autonomous systems," ArduPilot.org. Accessed: 2023. [Online]. Available: https://ardupilot.org/

[6] K. Cheng et al., "DTaint: Detecting the taint-style vulnerability in embedded device firmware," in *Proc. 48th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 430–441.

[7] D. She, Y. Chen, A. Shah, B. Ray, and S. Jana, "Neutaint: Efficient dynamic taint analysis with neural networks," in *Proc. IEEE Symp. Secur. Privacy (SP)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 1527–1543.

[8] V. Chibotaru, B. Bichsel, V. Raychev, and M. Vechev, "Scalable taint specification inference with big code," in *Proc. 40th ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI)*, New York, NY, USA: ACM, 2019, pp. 760–774.

[9] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and B. Baudry, "Test them all, is it worth it? Assessing configuration sampling on the JHipster web development stack," *Empirical Softw. Eng.*, vol. 24, no. 2, pp. 674–717, 2019.

[10] T. Kim et al., "RVFuzzer: Finding input validation bugs in robotic vehicles through control-guided testing," in *Proc. 28th USENIX Secur. Symp. (USENIX Secur.)*, Berkeley, CA, USA: USENIX Association, Aug. 2019, pp. 425–442.

[11] R. Han et al., "Control parameters considered harmful: Detecting range specification bugs in drone configuration modules via learning-guided search," in *Proc. Process. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2022, pp. 462–473.

[12] D. Whitley, "A genetic algorithm tutorial," *Statist. Comput.*, vol. 4, no. 2, pp. 65–85, 1994.

[13] DeviationVideo. *Deviation With Overshoot*. (2022). Accessed: 2023. [Online Video]. Available: https://youtu.be/imsBGaX-8ug

[14] DeviationVideo2. *Deviation With Flying Away*. (2022). Accessed: 2023. [Online Video]. Available: https://youtu.be/0buFPNhkLJc

[15] FlightFreezeVideo. *Flight Freeze*. (2022). Accessed: 2023. [Online Video]. Available: https://youtu.be/g79lsTp6H4g

[16] CrashVideo. *Crash*. (2022). Accessed: 2023. [Online Video]. Available: https://youtu.be/TZFcyl5d2mk

[17] PrivilegeEscalationVideo. *Post-Launch Privilege Escalation Leads to Crash*. (2022). Accessed: 2023. [Online Video]. Available: https://youtu.be/B_WULGY-Dbg

[18] ArduPilot Team, "SparkFun autonomous vehicle competition 2013," SparkFun Electronics. Accessed: 2023. [Online]. Available: https://avc.sparkfun.com/2013

[19] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with LSTM," *Neural Comput.*, vol. 12, no. 10, pp. 2451–2471, 2000.

[20] R. Akita, A. Yoshihara, T. Matsubara, and K. Uehara, "Deep learning for stock prediction using numerical and textual information," in *Proc. 15th IEEE/ACIS Int. Conf. Comput. Inf. Sci. (ICIS)*, Piscataway, NJ, USA: IEEE Press, 2016, pp. 1–6.

[21] F. Altché and A. de La Fortelle, "An LSTM network for highway trajectory prediction," in *Proc. 20th IEEE Int. Conf. Intell. Transp. Syst. (ITSC)*, Piscataway, NJ, USA: IEEE Press, 2017, pp. 353–359.

[22] S. Selvin, R. Vinayakumar, E. A. Gopalakrishnan, V. K. Menon, and K. P. Soman, "Stock price prediction using LSTM, RNN and CNN-sliding window model," in *Proc. Int. Conf. Adv. Comput., Commun. Inform. (ICACCI)*, Piscataway, NJ, USA: IEEE Press, 2017, pp. 1643–1647.

[23] Z. Wang and A. C. Bovik, "Mean squared error: Love it or leave it? A new look at signal fidelity measures," *IEEE Signal Process. Mag.*, vol. 26, no. 1, pp. 98–117, Jan. 2009.

[24] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN," *ACM Trans. Database Syst.*, vol. 42, no. 3, pp. 1–21, 2017.

[25] ArduPilot Development Team, "Autopilot hardware options," ArduPilot.org. Accessed: 2023. [Online]. Available: https://ardupilot.org/copter/docs/common-autopilots.html#closed-hardware

[26] L. Meier, P. Tanskanen, F. Fraundorfer, and M. Pollefeys, "PIXHAWK: A system for autonomous flight using onboard computer vision," in *Proc. IEEE Int. Conf. Robot. Automat. (ICRA)*, Piscataway, NJ, USA: IEEE Press, 2011, pp. 2992–2997.

[27] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "AirSim: High-fidelity visual and physical simulation for autonomous vehicles," in *Proc. Field Service Robot.*, 2017, pp. 621–635.

[28] DroneCode, "jMAVSim: A simple multirotor/quad simulator," Dronecode. Accessed: 2023. [Online]. Available: https://dev.px4.io/v1.10_noredirect/en/simulation/jmavsim.html

[29] H. Choi et al., "Detecting attacks against robotic vehicles: A control invariant approach," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA: ACM, 2018, pp. 801–816.

[30] G. Heredia, A. Ollero, M. Bejar, and R. Mahtani, "Sensor and actuator fault detection in small autonomous helicopters," *Mechatronics*, vol. 18, no. 2, pp. 90–99, 2008.

[31] I. Samy, I. Postlethwaite, and D. Gu, "Neural network based sensor validation scheme demonstrated on an unmanned air vehicle (UAV) model," in *Proc. 47th IEEE Conf. Decis. Control (CDC)*, Piscataway, NJ, USA: IEEE Press, 2008, pp. 1237–1242.

[32] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: Efficient fuzzing with neural program smoothing," in *Proc. IEEE*

Symp. Secur. Privacy (SP), Piscataway, NJ, USA: IEEE Press, 2019, pp. 803–817.

[33] G. Yan, J. Lu, Z. Shu, and Y. Kucuk, "ExploitMeter: Combining fuzzing with machine learning for automated evaluation of software exploitability," in *Proc. IEEE Symp. Privacy-Aware Comput. (PAC)*, Piscataway, NJ, USA: IEEE Press, 2017, pp. 164–175.

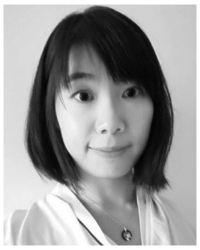
[34] Y. Chen, C. M. Poskitt, J. Sun, S. Adepu, and F. Zhang, "Learning-guided network fuzzing for testing cyber-physical system defences," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 962–973.

[35] V. Mnih et al., "Playing Atari with deep reinforcement learning," 2013, *arXiv:1312.5602*.

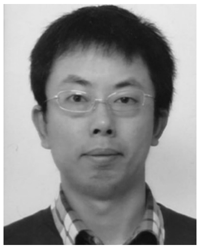
[36] K. Böttinger, P. Godefroid, and R. Singh, "Deep reinforcement fuzzing," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 116–122.

[37] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, 2018, pp. 95–105.

Ruidong Han received the B.Eng. degree in computer science from Northwest A&F University, in 2018, China and the Ph.D. degree from the School of Cyber Engineering, Xidian University. His research interests include IoT security, trusted computing, and intelligent system security.



Siqi Ma (Member, IEEE) received the Ph.D. degree in information system from Singapore Management University, in 2018. She is currently a Lecturer with the School of Engineering and Information Technology at the University of New South Wales. Her research interests are mobile security, web security, and IoT security.



Juanru Li (Member, IEEE) is the Director with the Group of Software Security In Progress (G.O.S.S.I.P), Shanghai Jiao Tong University. His research interests cover a wide array of systems and software security problems, with an emphasis on designing practical solutions and techniques that help computer systems stay secure.

Surya Nepal (Senior Member, IEEE) is a Senior Principal Research Scientist with CSIRO's Data61. He currently leads with the Distributed Systems Security Group comprising more than 30 research staff and more than 50 postgraduate students. His main research focus is in the development and implementation of technologies in the area of cybersecurity and privacy, and AI and cybersecurity. He is a member of the editorial boards of IEEE TRANSACTIONS ON SERVICE COMPUTING, *ACM Transactions on Internet Technology*, IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING.

David Lo (Fellow, IEEE) received the Ph.D. degree in computer science from the National University of Singapore, Singapore, in 2008. He is currently a fellow with Automated Software Engineering, ACM Distinguished Member, and Professor of Computer Science at Singapore Management University. His research interests include the intersection of software engineering and data science, encompassing socio-technical aspects, and analysis of different kinds of software artefacts, with the goal of improving software quality and developer productivity. His work has been published in premier and major conferences and journals in the area of software engineering, AI, and cybersecurity.

Zhuo Ma (Member, IEEE) received the Ph.D. degree in computer architecture from Xidian University, Xi'an, China, in 2010. Now, he is an Associate Professor with the School of Cyber Engineering, Xidian University. His research interests include cryptography, machine learning in cyber security, and Internet of Things security.

Jianfeng Ma (Member, IEEE) received the B.S. degree in computer science from Shaanxi Normal University, in 1982, the M.S. degree in computer science from Xidian University, in 1992, and the Ph.D. degree in computer science from Xidian University, in 1995. He is currently a Professor with the School of Computer Science and Technology, Xidian University. He has published over 150 journal and conference papers. His research interests include information security, cryptography, and network security.