UAVConfigFuzzer: Detecting Incorrect Configurations in Unmanned Aerial Vehicles via Setpoint Estimation Guided Fuzzing (Registered Report)

Yingnan Zhou
Nankai University
yingnan.zhou@mail.nankai.edu.cn

Yuhao Liu
Nankai University
yuhao.liu@mail.nankai.edu.cn

Hanfeng Zhang
Nankai University
zhanghf2048@gmail.com

Yan Jia*
Nankai University
jiay@nankai.edu.cn

Sihan Xu*
Nankai University
xusihan@nankai.edu.cn

Zhiyuan Jiang
National University of Defense Technology
jzy@nudt.edu.cn

Zheli Liu
Nankai University
liuzheli@nankai.edu.cn

***Abstract*—Flight control software for unmanned aerial vehicles (UAVs) offers numerous configuration parameters. However, their complexity raises the risk of incorrect configurations, leading to mission failures or crashes. Although fuzzing is effective for discovering software vulnerabilities, its application to UAVs configuration is hindered by the need to obtain physical states (e.g., position and altitude) from a time-consuming simulator. Furthermore, machine learning-based acceleration methods often suffer from limited generalizability due to their reliance on flight logs as training data. To address these challenges, we propose UAVConfigFuzzer, a novel fuzzing tool that accelerates configuration testing via setpoint estimation guided fuzzing. In flight control software, setpoints are the calculated target values that guide the UAV's movement based on configurations. UAVConfigFuzzer leverages the native setpoint generation module to generate setpoints, which serve as the estimated UAV's physical states to rapidly quantify the severity of UAV's anomalies. Guided by this efficient and accurate feedback, UAVConfigFuzzer steers the mutation process toward anomaly-inducing configurations without relying on simulators or extensive flight logs. We evaluate UAVConfigFuzzer on PX4, a widely used open-source UAV flight control software, the results demonstrate that the feedback achieves an average runtime of 27 milliseconds. The estimated states maintain high fidelity, with a mean position error below 6.92 cm and a velocity error below 0.13 m/s. Leveraging this rapid feedback, UAVConfigFuzzer detects 14 incorrect configurations. These issues were validated on real UAV hardware and have been acknowledged by the community maintainers for remediation.**

I. INTRODUCTION

Flight control software serves as the core system that governs the behavior of unmanned aerial vehicles (UAVs), determining their flight performance and ensuring safety [1]. In autopilot mode, UAVs follow a predefined flight mission uploaded by the user, with the flight control software autonomously performing missions such as takeoff and hovering [2]. To enhance the flexibility of autopilot mode, the flight control software offers users hundreds of configuration parameters, allowing customization of various aspects of autopilot performance, including cruise speed, yaw alignment, and flight trajectory smoothing [3]. However, due to the complexity of the configurations, inexperienced users can easily misconfigure them, leading to mission failures [4] and even crashes [5].

Fuzzing has become a prevalent testing method for detecting incorrect configurations in UAVs. In UAV fuzzing, most studies rely on simulator-based feedback [6], [7], [8], [9], [10], [11], [12] to collect the physical states of UAVs (e.g., positions and altitudes), to guide subsequent mutations and detect incorrect configurations. Despite its effectiveness, simulator-based UAVs fuzzing faces a significant efficiency bottleneck. For instance, simulating a flight mission usually takes minutes, resulting in a substantial time cost that severely limits the efficiency of the fuzzing process. To improve fuzzing efficiency, Han et al. [11] proposed *LGDFuzzer*, which employs a Long Short-Term Memory predictor [13] trained on historical flight logs to predict the physical states of UAVs. However, such a data-driven method inherently restricts its generalizability to new flight missions.

In this paper, we propose UAVConfigFuzzer to enhance the efficiency of UAVs configuration fuzzing. Instead of relying on simulators or collecting flight logs, UAVConfigFuzzer efficiently estimates UAV's physical states by leveraging *setpoints*. A setpoint represents the target physical state (e.g., position and velocity) generated by the flight control software to guide the UAV's movement. Since the controller is designed to control a UAV to follow these setpoints, any deviation of the generated setpoints from the user-defined mission implies that the UAV is in an anomalous state. Based on this insight, we reuse the native setpoint generation logic to rapidly estimate UAV's physical states, effectively bypassing simulation.

To achieve this, we design a tool named *setpoint generator*,

* Corresponding authors

Fuzzing Workshop (FUZZING) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-04-6
https://dx.doi.org/10.14722/fuzzing.2026.23009
www.ndss-symposium.org

which accepts the user's flight mission and configuration parameters as inputs and outputs the setpoint sequence. We conduct the setpoint generator by reusing the setpoint generation module from the flight control software. A key challenge lies in isolating it due to complex inter-module dependencies. We address this by leveraging the native build system to resolve these dependencies, enabling us to directly reuse the original source code of the setpoint generation logic. Built upon this novel setpoint-based UAV's physical state estimation tool, we propose UAVConfigFuzzer, a configuration fuzzing tool. UAVConfigFuzzer leverages the setpoint generator to produce setpoint sequences and quantifies the severity of three types of UAV's anomalies (i.e., rapid ascent/descent, deviation and interruption) by comparing generated setpoints against the user-defined mission. Guided by this feedback, UAVConfigFuzzer iteratively mutates configurations toward anomaly-inducing values to detect incorrect configurations, which are subsequently verified via a simulator. Finally, UAVConfigFuzzer refines the valid configuration bounds by excluding the confirmed incorrect values.

Evaluation results on the widely used flight control software, PX4, show that the simulator takes at least 28.47 seconds to estimate physical states, whereas our setpoint generator requires only 27 milliseconds. Crucially, this efficiency comes with high precision, yielding mean position and velocity errors of less than 6.92 cm and 0.13 m/s, respectively. Consequently, UAVConfigFuzzer achieves 100% accuracy in detecting three types of UAV's anomalies. Guided by this rapid feedback, UAVConfigFuzzer identified 14 incorrect configurations, including five new ones not found by the state-of-the-art fuzzer. We validate these findings on a real UAV equipped with Pixhawk 4 (PX4 v1.15.0) and report them to the community. The maintainers acknowledged the reported issues, confirming that they require either firmware patches or explicit documentation.

This paper makes the following contributions:

- We propose a novel physical state estimation method for fuzzing cyber-physical systems by reusing their native code, i.e., setpoint generation code from the flight control system. The outputs of this method serve as an oracle, enabling rapid and accurate detection of UAV's physical anomalies during fuzzing.
- Based on this idea, we implement UAVConfigFuzzer, a UAVs configuration fuzzing tool that leverages setpoint estimation as feedback, thereby eliminating the dependency on simulators or flight logs. Evaluation demonstrates that the setpoint generator estimates UAVs physical states in just 27 ms with mean errors below 6.92 cm in position and 0.13 m/s in velocity. Guided by this rapid and accurate feedback, UAVConfigFuzzer successfully identifies 14 incorrect configurations.
- We validate the detected incorrect configurations on a real UAV and report the findings to the community, and the maintainers have acknowledged them. Furthermore, we have open-sourced our tool[1] to facilitate future research.

[1]https://github.com/hapi2test/fuzz

II. Background

A. UAV Configuration Parameters

Configuration parameters are an essential component of flight control software, as they play a key role in ensuring flight stability and mission success. The UAV configuration parameters are typically key-value pairs and can be adjusted via the command line, configuration files, or the ground control station interface. However, flight control software often lacks proper validation for these configurations [14], [15], leading to the acceptance of invalid values and thereby increasing the risk of flight failures. For example, assigning excessively large values to roll and pitch controllers can cause instability in UAVs, potentially resulting in a crash [5]. Furthermore, the configuration parameters in flight control software exhibit complex correlations [16]. Configuration correlation refers to the dependencies or interactions between different configuration parameters, where changing one may affect the behavior or validity of another. Incorrect configurations can arise when these correlations are not satisfied. For instance, `MPC_XY_VEL_MAX` is a configuration parameter in PX4 that adjusts the maximum horizontal velocity. If `MPC_XY_VEL_MAX` is set to a value lower than `MPC_XY_CRUISE`, which indicates the default horizontal velocity during cruise, the UAV flies backward, performs erratic loops, and is unable to fly stably [4].

B. Trajectory Generator

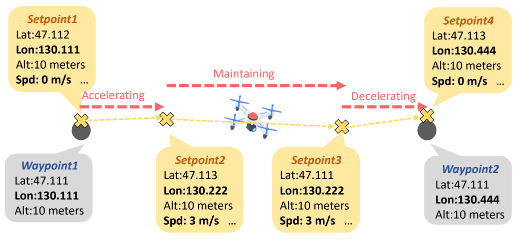

Fig. 1. Illustration of UAVs Trajectory Generation.

As shown in Figure 1, the user uploads a sequence of *waypoints*, collectively referred to as a *flight mission*. A waypoint defines a geographical coordinate (typically latitude, longitude, and altitude) that the UAV is required to reach. Upon receiving the mission, the flight control software employs a *trajectory generator* to construct a path connecting these waypoints, illustrated by the yellow path in the figure. The trajectory generator is a module responsible for calculating smooth motion paths that transition the UAV from its current position to the target destination. The outputs of this process are *setpoints*. Setpoints represent the discrete values sampled from the generated trajectory at specific timestamps. They serve as the target states (e.g., position, velocity, acceleration) that the UAV's controller strives to track.

The trajectory generator serves as a critical bridge between high-level planning and low-level controllers. It computes the flight path based on the flight mission and configuration parameters (e.g., velocity limits, acceleration constraints). Fundamentally, the setpoints output by the trajectory generator represent the target physical state of the UAVs. Consequently, the UAV's physical state can be estimated by analyzing these setpoints, eliminating the need for simulations. To acquire these setpoints, we can reuse the trajectory generator module in flight control software. This approach enables fuzzing of the configuration parameters and direct observation of their impact on the UAV's physical state through the generated setpoints.

C. UAV Anomalies

A UAV is considered to be in an anomalous state when there is a significant deviation from its predefined flight mission. Based on existing classifications of UAV's anomalies [17], [18], we design test oracles to detect three common types of UAV's anomalies. **Rapid Ascent/Descent:** UAVs are expected to maintain smooth and controlled speed changes during operation, as rapid fluctuations in vertical velocity, can lead to instability, loss of control, or even crashes. **Deviation:** A deviation anomaly occurs when the discrepancy between the actual flight trajectory and the mission trajectory exceeds a predefined threshold, potentially preventing the UAV from returning to the intended path. **Interruption:** In autopilot mode, UAVs must remain in motion during flight. If a UAV's position remains stationary for an extended period, with its movement distance falling below a predefined threshold, this condition is classified as an interruption anomaly. Such anomalies can hinder the UAV's ability to execute subsequent flight missions.

III. UAVConfigFuzzer

A. Overview

Figure 2 illustrates the workflow of UAVConfigFuzzer, which comprises three modules: *Constraint-based Mutation*, *Feedback Engine*, and *Validation & Refinement*. First, UAVConfigFuzzer mutates configurations based on extracted constraints (i.e., bound and correlation) via *1*-D and n-D mutation strategies. The *1*-D mutation mutates individual configuration one by one within the bounds. The n-D mutation mutates multiple correlated configurations. Subsequently, in the feedback engine, the setpoint generator takes the mutated configurations and flight mission as input to estimate setpoints. Based on these setpoints, UAVConfigFuzzer calculates fitness scores to detect three types of UAV's anomalies, i.e., rapid ascent/descent, deviation, and interruption. A high fitness score indicates that the configuration is more likely to be incorrect. Next, the fuzzing loop continues using the configurations with high fitness scores as seeds for the next round of mutation. Once the fuzzing loop terminates after a predefined number of iterations, it outputs a set of candidate incorrect configurations. Finally, in the validation & refinement module, UAVConfigFuzzer uses a simulator to verify the discovered incorrect configuration values. Confirmed incorrect values are excluded from the configuration bounds, and the remaining intervals are defined as the final valid bounds.

B. Setpoint Generator

We develop a tool named setpoint generator to generate setpoints based on a flight mission and configuration parameters. To ensure consistency with the flight control software, we directly reuse the trajectory generator module (i.e., PositionSmoothing) instead of re-implementing the generation logic. This method avoids the challenges posed by the flight control software's complex dependencies, which often prevent static extraction or logic rewriting from compiling successfully.

The detailed procedure of the setpoint generator is outlined in Algorithm 1. First, we leverage the native build system (CMake) to resolve complex dependencies, enabling us to link the setpoint generation logic into a standalone executable. The algorithm accepts a configuration vector C and a flight mission M_{geo} as inputs. The mission is defined as an ordered sequence of N waypoints in the geodetic coordinate system (latitude, longitude, altitude). In the initialization phase, the algorithm transforms M_{geo} into the local North-East-Down (NED) frame, denoted as M (line 2). The Device Under Test (DUT) is defined as the PositionSmoothing module, which encapsulates the core trajectory generation algorithms. To enable fine-grained control over protected internal members, we implement an intrusive wrapper that inherits from the DUT, granting the algorithm direct write access to inject the configuration vector C (line 3, line 4). During the main execution loop, the algorithm constructs a $M_{triplet}$ (previous, current, and next waypoints) to capture the local navigation context. The DUT then generates the setpoint s_{t+1} for the next time step Δt (line 10). Crucially, if the generated setpoint exhibits numerical instability, the process terminates immediately, returning the partial setpoint sequence. Otherwise, under the assumption of an ideal environment, the generated setpoint is directly used as the next UAV state x_{t+1} in the subsequent iteration (line 15). Next, the waypoint switches to the target index when the UAV enters the acceptance radius R_{accept}. Finally, the complete setpoints sequence S is returned.

Algorithm 1 Setpoint Generator

```
Require: Configuration vector $C$
Require: Flight mission $M_{geo} = \{m_0, m_1, \dots, m_N\}$
Ensure: Setpoints sequence $S$
 1: Initialization:
 2: $M \leftarrow$ MapFunction($\mathcal{M}_{geo}$, Ref $= m_0$)
 3: $DUT \leftarrow$ IntrusiveWrapper(PositionSmoothing)
 4: $DUT$.injectConfig($C$)
 5: $x_t \leftarrow M[0]$
 6: $idx \leftarrow 1$
 7: $S \leftarrow \emptyset$
 8: while $t < T_{\max}$ and $idx < N$ do
 9:     $W_{triplet} \leftarrow \{M[idx-1], M[idx], M[idx+1]\}$
10:     $s_{t+1} \leftarrow DUT$.generateSetpoints($x_t, W_{triplet}, \Delta t$)
11:     $S \leftarrow S \cup \{s_{t+1}\}$
12:     if $\neg$IsFinite($s_{t+1}$) then
13:         return $S$
14:     end if
15:     $x_{t+1} \leftarrow s_{t+1}$
16:     if $\|x_{t+1}.\text{pos} - M[idx]\| < R_{\text{accept}}$ then
17:         $idx \leftarrow idx + 1$
18:     end if
19:     $t \leftarrow t + \Delta t$
20: end while
21: return $S$
```

Note that the setpoint generator focuses on logic verification under ideal environmental conditions (i.e., zero wind disturbances, no air turbulence, and an obstacle-free space). By

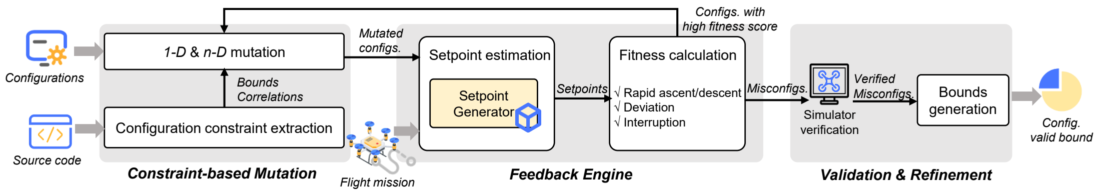

Fig. 2. Workflow of UAVConfigFuzzer.

eliminating external environmental factors, we ensure that any detected anomalies are attributed to incorrect configurations rather than external factors. Furthermore, the setpoint generator features a flexible architecture. Specifically, the setpoint generator can be extended to include environmental factors by adding a disturbance model to the UAVs state update at line 15, formulated as $x_{t+1} \leftarrow s_{t+1} + ENV$, to simulate the influence of the environment on the UAV's physical states.

C. Setpoint Estimation Guided Fuzzing

This section describes the three core components of the fuzzing loop, including constraint extraction to define the mutation scope, *1*-D and n-D mutations to generate configurations, and fitness evaluation to guide the search toward potential anomalies.

1) Configuration Constraint Extraction: To mitigate the vast search space in configuration fuzzing, UAVConfigFuzzer performs static analysis to extract configuration constraints, namely bounds and correlations. By utilizing these constraints to define the mutation scope, UAVConfigFuzzer effectively filters out-of-bound values and incompatible combinations.

UAVConfigFuzzer begins by identifying configuration read sites and mapping configuration names from the user guide to their corresponding data structures in the code. To accurately model data propagation, UAVConfigFuzzer applies Andersen's analysis [19] to resolve pointer aliases, ensuring that all aliased variables are included in the configuration variable set. Subsequently, a Mod-Ref analysis is performed to pinpoint where these variables are modified or referenced. Based on these analyses, UAVConfigFuzzer constructs a value-flow graph that captures the flow and dependencies of the configurations.

With the value-flow graph constructed, UAVConfigFuzzer traverses the graph to identify two constraint types, i.e., bounds and correlations. The bounds are the minimum and maximum values of a configuration parameter, and the correlations represent the constraints between two configuration parameters. Bounds are identified by detecting calls to specific boundary functions. UAVConfigFuzzer looks for patterns like $\langle \textit{boundCons} \rangle := \langle \textit{boundFunc} \rangle\,(\langle \textit{paramRef} \rangle, \langle \textit{arg1} \rangle, \dots)$, where *boundFunc* represents the functions that limit the values of a configuration variable within a range, such as *min()*, *max()*, and *interpolate()*. Correlations between configurations are typically found in conditional statements that use comparison operators. UAVConfigFuzzer searches for patterns like $\langle \textit{corrCons} \rangle := \langle \textit{compOper} \rangle\,(\langle \textit{paramRef} \rangle, \langle \textit{arg} \rangle)$, where *compOper* represents comparison operators such as $<$ and $>$, and *arg* can be a configuration parameter or a constant.

This entire static analysis process is performed only once before the fuzzing begins and outputs the bounds and correlations of each configuration parameter.

2) Mutation Strategy: 1-D mutation focuses on identifying the safe bound for each individual configuration parameter. UAVConfigFuzzer mutates one target configuration at a time while maintaining others at their default values. UAVConfigFuzzer initializes N configuration sets, each composed of the target configuration and others with default values, by randomly sampling values within the bounds extracted in Section III-C1. Subsequently, UAVConfigFuzzer employs the fitness function to calculate scores for each configuration set. After the calculation, the configuration sets with the top K fitness scores are selected as seeds for the next mutation. UAVConfigFuzzer performs a localized search by generating N new target values within a mutation range of $\pm 50\%$ around these seeds. This process iterates until a predefined number of mutations is reached. Finally, high-fitness configurations identified during fuzzing are considered candidates for unsafe values, which are verified via a simulator. Verified unsafe values are then excluded from the initial bound, and the remaining intervals are marked as the single-safe bound.

n-D mutation aims to detect anomalies arising from the correlations between configurations. Since directly exploring all combinations leads to a combinatorial explosion, UAVConfigFuzzer leverages the correlation constraints identified in Section III-C1 to prune the search space. Specifically, UAVConfigFuzzer mutates the target configuration by randomly sampling N values within its refined single-safe bounds, while simultaneously assigning critical values (i.e., maximum, minimum, and default) to the correlated configurations. This strategy generates $3N$ configuration sets per iteration. Similar to the *1*-D process, UAVConfigFuzzer calculates fitness scores, ranks, and mutates these sets iteratively. Finally, the unsafe values discovered during this phase are verified by the simulator. These validated unsafe values are subtracted from the single-safe bounds to derive final valid bounds for each configuration.

3) Fitness Calculation: The fitness calculation process compares the estimated setpoints from the setpoint generator against the user-defined flight mission to quantify the severity of three types of UAV's anomalies via specific fitness scores. A higher fitness score indicates a greater likelihood that the UAV is experiencing an anomaly. Following the classification of UAV's anomalies in Section II-C, we develop three methods to calculate their fitness scores. *Rapid Ascent/Descent*: This anomaly is characterized by abrupt changes in vertical velocity, corresponding to excessive acceleration along the Z-axis. The fitness score is defined as the maximum absolute Z-axis acceleration observed in the setpoint sequence. *Deviation*: This anomaly manifests when the generated setpoints significantly diverge from the user-defined flight path. UAVConfigFuzzer models the flight mission as line segments connecting consecutive waypoints. It then calculates the perpendicular distance from each setpoint to the corresponding mission segment. The fitness score is defined as the maximum deviation distance observed across the entire setpoint sequence. *Interruption*: This anomaly captures scenarios where the UAV gets stuck (i.e., velocity drops to near zero) or fail to complete its mission. The fitness function first verifies whether the setpoint sequence is complete. If the sequence is incomplete, the mission is considered interrupted, and the fitness score is assigned an infinite value. Otherwise, the fitness score is defined as the inverse of the minimum 3D total velocity $V_{total=\sqrt{v_x^2+v_y^2+v_z^2}}$ in the setpoint sequence.

IV. Preliminary Evaluation

This section presents the preliminary evaluation of UAVConfigFuzzer and addresses three research questions.

- **RQ1: Efficiency.** How does the setpoint generator enhance the efficiency of feedback?
- **RQ2: Accuracy.** How accurate are the components of UAVConfigFuzzer, specifically in estimating setpoints and extracting configuration constraints?
- **RQ3: Effectiveness.** How effective is UAVConfigFuzzer in detecting incorrect configurations and valid bounds?

A. Implementation

We conduct our experiments using Gazebo [20] as the simulator and QGroundControl [21] as the ground control station, with PX4 1.15.0 [22] as the target flight control software. All experiments are performed on a machine with an Intel Core i7-14700KF CPU and 32GB memory running Ubuntu 20.04. We further validate the detected anomalies on a real AMOVLab P450 quadcopter [23] equipped with Pixhawk 4 [24] and PX4 1.15.0, and provide three demonstration videos [25], [26], [27]. In the implementation, the setpoint output is restricted to the three-dimensional position (x, y, z) and velocity (v_x, v_y, v_z), which suffice for anomaly detection. The hyperparameter settings in fuzzing module were guided by common practices [11], [28] and informed by preliminary experiments. The minimum mutation step size was determined by the increment specified in the user guide. Furthermore, the number of random values (N) was set to 100, the number of selections (K) was set to 5, and the number of iterations was fixed at 200.

B. RQ1: Efficiency

To evaluate the time efficiency of the setpoint generator, we benchmarked its runtime against Gazebo on 50 flight missions with 10 randomly distributed waypoints each. Both systems executed the missions under default configurations, and Gazebo was optimized using headless mode and the maximum `PX4_SIM_SPEED_FACTOR` (a configuration controlling the simulation speed relative to real-time) to minimize simulation time, after which we measured the average runtime.

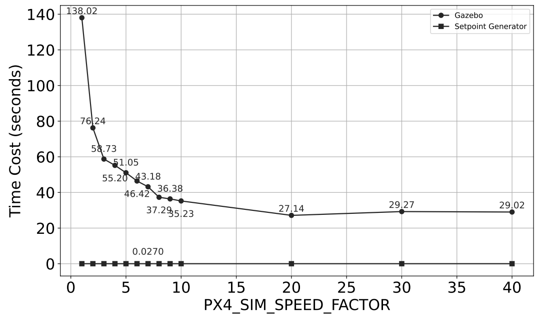

Fig. 3. The Average Runtime of Gazebo and the Setpoint Generator.

As illustrated in Figure 3, the runtime of the simulator decreases as the `PX4_SIM_SPEED_FACTOR` increases. The runtime plateaus at an average of 28.47 seconds once the speed factor exceeds 20. In contrast, the setpoint generator consistently achieves an average runtime of 27 milliseconds, yielding a speedup of more than $1{,}000\times$ over simulation-based execution. This efficiency stems from the architectural design of the setpoint generator, which bypasses the computational overhead of full physics simulation. As detailed in Section III-B, rather than performing a step-by-step physics simulation, the setpoint generator directly leverages the trajectory generation algorithm embedded within the flight control software. This allows it to compute the UAV's physical states rapidly, achieving millisecond-level estimation.

C. RQ2: Accuracy

1) The Setpoint Generator: To validate the fidelity of the setpoint generator, we benchmarked the setpoints it produces against those generated by the native flight control software. We employed a ten waypoint flight mission executed in the PX4 with default configurations. The setpoints generated by the flight control software, serving as the ground truth, were extracted from the flight logs. We then processed the same mission and configurations using the setpoint generator and compared the estimated setpoints against the ground truth.

As illustrated in Figure 4, the setpoints generated by the setpoint generator align closely with the ground truth. While the overall trends are consistent, minor deviations are observed. Specifically, the ground truth data exhibits slight fluctuations. These deviations are attributable to the physical simulation, where the control layer corrects for disturbances, causing the UAVs to react with a slight delay and constantly adjust their states. In contrast, the setpoint generator operates under an ideal feedback assumption. By eliminating the control layer, the setpoint generator produces noise-free setpoints.

To evaluate the generalizability of the setpoint generator across diverse flight scenarios, we conducted a statistical evaluation comprising 100 randomized test cases. Each test case

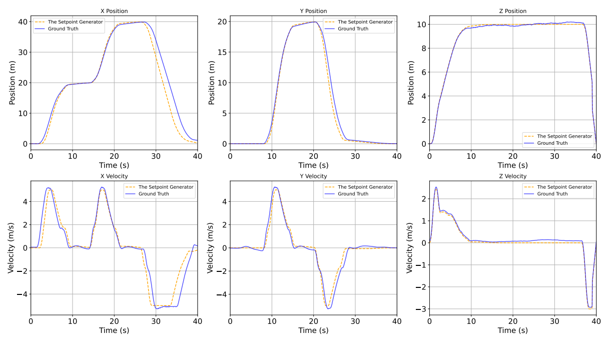

Fig. 4. Comparison of Interpolated Setpoint Trajectories between the Setpoint Generator and Ground Truth Generated by Simulator.

consists of a randomly generated mission with ten waypoints and a configuration set. To ensure comprehensive coverage of the configuration space, we employed a sampling strategy for configuration generation, i.e., 50% of configurations were assigned default values, 10% were set to the minimum bound, 10% to the maximum bound, and the remaining 30% were randomly sampled within the valid range. The ground truth was established using the same methodology as the fidelity validation. We quantified the discrepancy between the setpoints generated by the setpoint generator and the ground truth using two metrics, i.e., the Mean Absolute Error (MAE) and Standard Deviation (Std).

TABLE I
Comparison of Mean Absolute Error and Standard Deviation between the Setpoint Generator and Ground Truth.

Metrics	**Position Error (cm)**			**Velocity Error (m/s)**		
	X	**Y**	**Z**	**X**	**Y**	**Z**
MAE	6.92	5.47	5.85	0.13	0.07	0.10
Std	5.02	5.32	3.99	0.22	0.31	0.18

As shown in Table I, the discrepancy of setpoints between the setpoint generator and the ground truth is negligible. In terms of position, the largest MAE occurs on the X-axis, with a value of only 6.92 cm. Regarding velocity, the MAEs remain below 0.13 m/s across all axes. Furthermore, the low standard deviations demonstrate the robustness of the setpoint generator, ensuring consistent performance across diverse missions and configurations.

2) Configuration Constraint Extraction: To evaluate the accuracy of the configuration constraint extraction module, we addressed the absence of a ground truth by implementing a two-step verification process. First, we established a ground truth through manual inspection of the source code. To ensure accuracy, this manual cross-validation was performed independently by two authors. Second, we compared the extracted constraints against the official user guide to assess alignment with the documentation.

As detailed in Table II, the constraint extraction module successfully identified a total of 19 configuration bounds and 16 correlation pairs. Manual verification against the source code confirmed that our method achieved 100% extraction accuracy. A comparison between the user guide and the implementation revealed significant discrepancies. Specifically, only 6 lower and 6 upper bounds in the user guide are consistent with the source code. For correlations, the user guide lists only 6 pairs. The high accuracy against the source code demonstrates the effectiveness and reliability of the static analysis method in extracting configuration constraints, which provides an accurate mutation scope for fuzzing. However, the discrepancy between our results and the user guide reveals a critical gap between implementation and documentation. For instance, the configuration `MPC_ACC_HOR` is constrained to [1.0, 15.0] in the code, but the guide suggests [2.0, 15.0]. Similarly, `MPC_Z_VEL_MAX_UP` is coded as $[0, +\infty)$ but documented as [0.5, 0.8]. These differences arise because the user guide often provides recommended values based on flight experience, not the actual hard-coded constraints enforced by the source code [29]. Furthermore, the user guide is incomplete, omitting 62.5% of the correlations identified in the source code, such as the constraint that `MPC_XY_CRUISE` must be smaller than `MPC_XY_VEL_MAX`. This inconsistency between implementation and documentation can mislead developers and users, leading to misconfigurations.

TABLE II
Evaluation of Extracted Constraints against Source Code and User Guide.

	Manual	**User Guide**		
		Entire	Min	Max
Bound	19/19	2/19	6/19	6/19
	Manual	**User Guide**		
Correlation	16/16		6/16	

D. RQ3: Effectiveness

To assess the anomaly identification capability of fitness calculation, we employed a dataset of 50 test cases, each consisting of a fixed flight mission and randomized configurations. Ground truth was established via Gazebo, where we observed 14 anomalous instances and 36 nominal flights. Specifically, the anomalies included 5 cases of rapid ascent/descent, 4 deviation, and 5 interruption. We then processed the same inputs into the fitness calculation process to compute the fitness scores for each configuration set. The results revealed that the fitness scores for anomalous configuration parameters were consistently higher than those for nominal flights. When validated against the ground truth, the fitness function successfully identified all 14 anomalies. This result confirms that the fitness calculation process serves as a reliable module for identifying UAV's anomalies to guide the mutation toward anomaly-inducing configurations.

To evaluate the effectiveness of UAVConfigFuzzer in detecting incorrect configurations, we compared its ability against RVFuzzer [6], a state-of-the-art fuzzer that relies on a binary search method and simulator-based validation.

As shown in Table III, UAVConfigFuzzer successfully detects a total of 14 incorrect configurations. Crucially, five of

TABLE III
INCORRECT CONFIGURATIONS IDENTIFIED BY UAVCONFIGFUZZER.

Configuration	**RVFuzzer**			**UAVConfigFuzzer**		
	R.	D.	I.	R.	D.	I.
MPC_TILTMAX_AIR		✓	✓		✓	✓
MPC_ACC_DOWN_MAX				✓		
MPC_XY_VEL_ALL				✓	✓	✓
MPC_XY_CRUISE		✓			✓	
MPC_XY_VEL_MAX		✓	✓		✓	✓
MPC_Z_VEL_ALL				✓	✓	
MPC_Z_V_AUTO_UP						✓
MPC_Z_VEL_MAX_UP	✓	✓	✓	✓	✓	✓
MPC_Z_V_AUTO_DN						✓
MPC_Z_VEL_MAX_DN		✓	✓	✓	✓	✓
MPC_TKO_SPEED			✓			✓
MPC_LAND_SPEED			✓			✓
MPC_TILTMAX_LND					✓	✓
MPC_YAWRAUTO_MAX	✓		✓	✓		✓

# R. shorts for 'Rapid ascent/descent', D. shorts for 'Deviation', I. shorts for 'Interruption'. ✓ indicates newly identified compare to RVFuzzer.

these are newly discovered misconfigurations that RVFuzzer failed to detect. As detailed in Section III-C2, UAVConfig-Fuzzer employs a random sampling strategy for initialization, enabling a global search across the entire configuration space. This is critical for identifying misconfigurations with discontinuous bounds. For example, UAVConfigFuzzer identifies two such cases, i.e., `MPC_XY_VEL_ALL` and `MPC_Z_VEL_ALL`. The user guide for `MPC_XY_VEL_ALL` specifies a valid range of [–20, 20], but UAVConfigFuzzer discovers that setting the value to precisely 0 triggers an interruption anomaly. In contrast, RVFuzzer failed to detect these two incorrect configurations because it applies a binary search strategy that assumes the valid configuration bound is continuous. Furthermore, UAVConfigFuzzer performs mutations based on configuration bounds extracted directly from the source code, whereas RVFuzzer relies on the bounds specified in the user guide. As our evaluation reveals that the user guide is often incomplete, which limits RVFuzzer's search space. By leveraging the source code as the ground truth, UAVConfigFuzzer explores a broader and more realistic configuration space, enabling the discovery of incorrect configurations that documentation-dependent methods fail to detect.

1) Ablation study of Mutation Strategy: To evaluate the effectiveness of the proposed mutation strategies, we conducted an ablation study to quantify their contributions. We used the same feedback engine module and the same number of mutation iterations for all mutation strategies. Specifically, we benchmarked our proposed *1-D* and *n-D mutations* against the *genetic algorithm* (used in LGDFuzzer [11]) and *binary search* (used in RVFuzzer [6]). The genetic algorithm operates on a set of configurations, and before crossover and selection, it mutates all configuration values simultaneously. The binary search strategy, instead of random mutation, systematically generates the next candidate value by calculating the midpoint of the current search interval, iteratively narrowing the range. Through this comparative analysis, we were able to quantify the capability of mutation strategies in generating valid configuration bounds.

As shown in Figure 5, the effectiveness of the four mutation strategies in generating valid bounds varies significantly. Compared to the baseline *1*-D mutation, the n-D strategy identifies more invalid values, resulting in narrower valid bounds for 26.3% of the configurations. The genetic algorithm produces even narrower valid bounds for 89.5% of the configurations. Notably, both the binary search and genetic algorithm strategies fail to detect two discontinuous valid bounds that were correctly identified by the *1*-D mutation. The performance of the n-D strategy stems from its ability to capture configuration correlations, allowing it to derive more precise bounds. For instance, it correctly identifies that the upper bound of `MPC_JERK_AUTO` is constrained by `MPC_JERK_MAX`, thereby explicitly flagging incompatible values as invalid. A critical limitation of the genetic algorithm is its coarse-grained mutation strategy. Since the algorithm randomly mutates multiple configurations, it lacks the granularity to pinpoint the root cause of misconfiguration. Consequently, a single invalid configuration value can cause an entire configuration set to be rejected. This erroneously penalizes correct configurations, leading to overly conservative bounds. As for the binary search, it relies on the assumption that the valid range of configuration parameters is continuous. This assumption caused it to miss the discontinuous valid intervals of `MPC_XY_VEL_ALL` and `MPC_Z_VEL_ALL`.

E. Case Study

To demonstrate UAVConfigFuzzer's capability in detecting incorrect configurations, we present a case study involving two configurations, i.e., `MPC_XY_VEL_ALL` is the overall limit for horizontal velocity, and `MPC_XY_VEL_MAX` is the maximum horizontal velocity. According to the official user guide, `MPC_XY_VEL_ALL` has a valid range of $[-20, 20]$. UAVConfigFuzzer reveals that setting `MPC_XY_VEL_ALL` to 0 causes the UAV to become stuck, while nearby values lead to normal operation, indicating a singular value that should be excluded from the valid bound. The root cause is shown in Figure 6. When `MPC_XY_VEL_ALL` is set to 0, it overrides `MPC_XY_VEL_MAX` and constrains the maximum horizontal velocity to zero via the `setVelocityLimits()` function. Using the *1*-D mutation strategy, UAVConfigFuzzer correctly identifies this discontinuity and refines the valid range of `MPC_XY_VEL_ALL` to $[-20, -1] \cup [1, 20]$.

V. FUTURE EVALUATION PLAN

In the second stage of this work, we plan to conduct two additional evaluations. First, we will add an ablation study on the mutation strategy by introducing a random mutation baseline to isolate the contribution of the proposed *1*-D and n-D mutations. We will sample configuration values using random mutation within the extracted bounds and evaluate them using the same fitness function and feedback engine under the same mutation budget. We will compare the quality of the derived valid configuration bounds produced by random

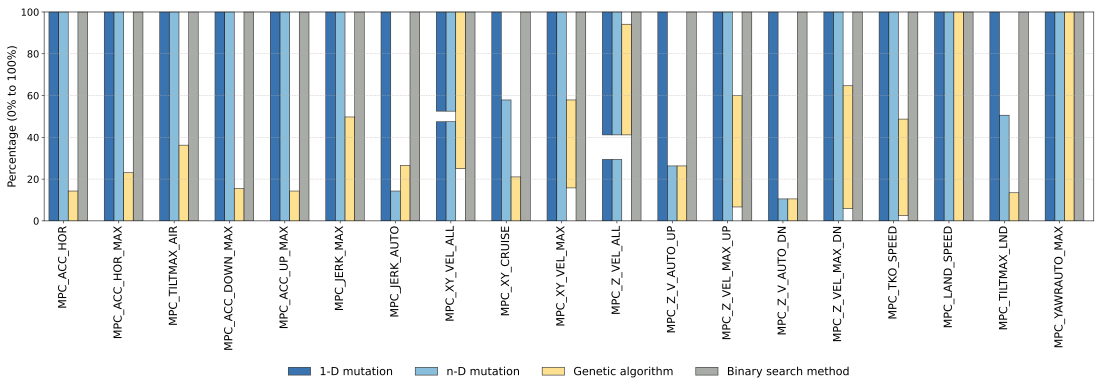

Fig. 5. Valid Configuration Bounds Derived by Different Mutation Strategies, Normalized to the Specified Value ranges. The Colored Bars Represent Valid Bounds, While the Blank Areas Are Invalid Bounds.

```
if (_param_mpc_xy_vel_all.get() >= 0.f) {
    float xy_vel = _param_mpc_xy_vel_all.get();
    num_changed += _param_mpc_xy_vel_max.commit_no_notification(xy_vel);
}
...
_control.setVelocityLimits(_param_mpc_xy_vel_max.get(), ...);
```

Fig. 6. Code snippet illustrating the correlations between `MPC_XY_VEL_ALL` and `MPC_XY_VEL_MAX`

mutation with those produced by our proposed mutation strategies. Second, we will conduct an in-depth case study on the anomaly cases reported by UAVConfigFuzzer, including both single-parameter misconfigurations and anomalies induced by configuration correlations. For each case, we will analyze the root causes and report the findings to the community, providing actionable insights for both UAV users and flight control software developers.

VI. Related Works

Incorrect configurations frequently lead to errors in robotic vehicles [30], [31], which has made fuzzing an important testing technique for detecting such issues [32], [33], [34], [35], [36], [11], [18], [37]. In the *initial phase*, accurately defining the mutation scope is critical. While PGFuzz [7] derives policies from user guides, our evaluation (Section IV-C) reveals that inconsistencies between implementation and documentation result in an incomplete mutation space. UAVConfigFuzzer addresses this by extracting constraints directly from the source code, achieving superior accuracy. Regarding the *mutation strategy*, tools like ConfVE [37] and RouthSearch [12] introduce optimization algorithms (e.g., genetic algorithms or stability criteria) to improve efficiency. However, they overlook configuration correlations, which leads to blind mutations and an exponential search space filled with invalid combinations. In contrast, UAVConfigFuzzer leverages these correlations to prune the search space, effectively reducing the complexity to linear $O(n)$. Finally, regarding the *feedback loop*, unlike existing methods limited by time-consuming simulations [8], [6], [38] or training data dependence [11], [39], UAVConfigFuzzer utilizes native setpoint logic to estimate UAV's physical states, thereby providing rapid and data-free feedback. This significantly improves efficiency and enhances generalizability.

VII. Conclusion

This paper addresses the critical challenge of inefficient feedback mechanisms that hinder UAVs configuration fuzzing. To overcome simulation efficiency bottlenecks and the data dependency of machine learning-based predictors, we propose a method to rapidly estimate UAV's physical states by reusing the native setpoint generation logic from the flight control software. This method leverages setpoints to quantify the severity of UAV's anomalies without relying on simulators or extensive training data. We implemented this method by developing a fuzzing tool named UAVConfigFuzzer. Guided by the setpoint estimation feedback, UAVConfigFuzzer iteratively mutates configurations toward anomaly-inducing values. Our evaluation demonstrates that the setpoint generation process achieves an average runtime of 27 ms while maintaining high fidelity in estimating UAV's physical states, with a mean position error below 6.92 cm and a velocity error below 0.13 m/s. Finally, UAVConfigFuzzer successfully detects 14 incorrect configurations, which are validated on real UAV hardware and reported to the community.

VIII. Revision Requirements

In the revision of the registered report, we plan to provide more fine-grained explanations of the scope of incorrect configurations targeted in this paper by refining the descriptions in the Introduction and Background sections. Then, we will further clarify the generalizability of UAVConfigFuzzer by adding a dedicated discussion on the applicability and limitations of the proposed method.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This work was funded by the National Natural Science Foundation of China (62572258, 62402509), the Fundamental Research Funds for the Central Universities, Nankai University (079-63253221, 070-63251116), and National Natural Science Foundation of Tianjin, China (25JCQNJC01380).

References

[1] Z. Zuo, C. Liu, Q.-L. Han, and J. Song, "Unmanned aerial vehicles: Control methods and future challenges," *IEEE/CAA Journal of Automatica Sinica*, vol. 9, no. 4, pp. 601–614, 2022.

[2] E. Ebeid, M. Skriver, K. H. Terkildsen, K. Jensen, and U. P. Schultz, "A survey of open-source uav flight controllers and flight simulators," *Microprocessors and Microsystems*, vol. 61, pp. 11–20, 2018.

[3] S. Serebryansky and K. Nastas, "Configuration management for unmanned aircraft basic systems at the different stages of its lifecycle," in *2021 14th International Conference Management of large-scale system development (MLSD)*. IEEE, 2021, pp. 1–5.

[4] M. L. maneuvers when MPC_XY_CRUISE over MPC_XY_VEL_MAX. (2019) Mc - loops maneuvers when mpc_xy_cruise over mpc_xy_vel_max. Https://github.com/PX4/PX4-Autopilot/issues/11420.

[5] A. flight and C. A. hold mode. (2023) Aggressive flight and crash: Alt hold mode. Https://discuss.ardupilot.org/t/aggressive-flight-and-crash-alt-hold-mode/98640.

[6] T. Kim, C. H. Kim, J. Rhee, F. Fei, Z. Tu, G. Walkup, X. Zhang, X. Deng, and D. Xu, "{RVFuzzer}: Finding input validation bugs in robotic vehicles through {Control-Guided} testing," in *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 2019, pp. 425–442.

[7] H. Kim, M. O. Ozmen, A. Bianchi, Z. B. Celik, and D. Xu, "Pgfuzz: Policy-guided fuzzing for robotic vehicles." in *NDSS*. ISOC, 2021.

[8] C. Jung, A. Ahad, J. Jung, S. Elbaum, and Y. Kwon, "Swarmbug: debugging configuration bugs in swarm robotics," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 868–880.

[9] S. Kim, M. Liu, J. J. Rhee, Y. Jeon, Y. Kwon, and C. H. Kim, "Drivefuzz: Discovering autonomous driving bugs through driving quality-guided fuzzing," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2022, pp. 1753–1767.

[10] S. Kim and T. Kim, "Robofuzz: fuzzing robotic systems over robot operating system (ros) for finding correctness bugs," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2022, pp. 447–458.

[11] R. Han, C. Yang, S. Ma, J. Ma, C. Sun, J. Li, and E. Bertino, "Control parameters considered harmful: Detecting range specification bugs in drone configuration modules via learning-guided search," in *Proceedings of the 44th International Conference on Software Engineering*. ACM, 2022, pp. 462–473.

[12] S. Wang, Z. Dong, H. Li, L. Shen, X. Peng, and D. She, "Routhsearch: Inferring pid parameter specification for flight control program by coordinate search," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 640–662, 2025.

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

[14] E. valid values of configuration parameters. (2021) Enforcing valid values of configuration parameters. Https://discuss.px4.io/t/enforcing-valid-values-of-configuration-parameters/23346.

[15] S. Purandare, U. Sinha, M. N. Al Islam, J. Cleland-Huang, and M. B. Cohen, "Self-adaptive mechanisms for misconfigurations in small uncrewed aerial systems," in *2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2023, pp. 169–180.

[16] S. oscillation on position hold. (2023) Slow oscillation on position hold. Https://discuss.px4.io/t/slow-oscillation-on-position-hold/31968.

[17] L. Li, "Anomaly detection in airline routine operations using flight data recorder data," *massachusetts institute of technology*, 2013.

[18] D. Wang, S. Li, G. Xiao, Y. Liu, Y. Sui, P. He, and M. R. Lyu, "An exploratory investigation of log anomalies in unmanned aerial vehicles," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, 2024, pp. 1–13.

[19] L. O. Andersen, "Program analysis and specialization for the c programming language," *Addison Wesley*, 1994.

[20] Gazebo. (2025) Gazebo. Https://gazebosim.org/home.

[21] Q. Intuitive and P. G. C. S. for the MAVLink protocol. (2025) Qgroundcontrol intuitive and powerful ground control station for the mavlink protocol. Https://qgroundcontrol.com/.

[22] O. S. A. F. D. P. Autopilot. (2025) Open source autopilot for drones - px4 autopilot. Https://px4.io/.

[23] A. Prometheus450. (2025) Amovlab prometheus450. [Online]. Available: https://wiki.amovlab.com/public/prometheuswiki/

[24] H. P. 4. (2025) Holybro pixhawk 4. Https://docs.px4.io/main/en/flight_controller/pixhawk4.html.

[25] demo video misconfiguration Rapid Ascent/Descent-crash. (2024) demo video misconfiguration-rapid ascent/descent-crash. Https://youtu.be/4rUY63GKm9c.

[26] demo video misconfiguration deviation. (2024) demo video misconfiguration-deviation. Https://youtu.be/-i2bFYIB7O0.

[27] demo video misconfiguration interruption. (2024) demo video misconfiguration-interruption. Https://youtu.be/MfE_ELEV_GQ.

[28] R. Han, S. Ma, J. Li, S. Nepal, D. Lo, Z. Ma, and J. Ma, "Range specification bug detection in flight control system through fuzzing," *IEEE Transactions on Software Engineering*, 2024.

[29] P.-A. T. angle parameter range check. (2019) Px4-autopilot#11473: Tilting angle parameter range check. [Online]. Available: https://github.com/PX4/PX4-Autopilot/issues/11473

[30] J. Garcia, Y. Feng, J. Shen, S. Almanee, Y. Xia, Chen, and Q. Alfred, "A comprehensive study of autonomous vehicle bugs," in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 385–396.

[31] X. Song, Y. Li, Z. Dong, S. Liu, J. Cao, and X. Peng, "An empirical study on fault diagnosis in robotic systems," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2023, pp. 207–219.

[32] V. K. Malviya, W. Minn, L. K. Shar, and L. Jiang, "Fuzzing drones for anomaly detection: A systematic literature review," *Computers & Security*, p. 104157, 2024.

[33] Honggfuzz. (2023) Honggfuzz. Https://llvm.org/docs/LibFuzzer.html.

[34] A. F. Lop. (2023) American fuzzy lop. Https://lcamtuf.coredump.cx/afl/.

[35] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16. ISOC, 2016, pp. 1–16.

[36] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2329–2344.

[37] Y. Chen, Y. Huai, S. Li, C. Hong, and J. Garcia, "Misconfiguration software testing for failure emergence in autonomous driving systems," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1913–1936, 2024.

[38] S. Feng, Y. Ye, Q. Shi, Z. Cheng, X. Xu, S. Cheng, H. Choi, and X. Zhang, "Rocas: Root cause analysis of autonomous driving accidents via cyber-physical co-mutation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1620–1632.

[39] S. Silalahi, T. Ahmad, H. Studiawan, E. Anthi, and L. Williams, "Severity-oriented multiclass drone flight logs anomaly detection," *IEEE Access*, 2024.