ADGFUZZ: Assignment Dependency-Guided Fuzzing for Robotic Vehicles

Yuncheng Wang[†‡*], Yaowen Zheng[†‡*], Puzhuo Liu[§¶], Dongliang Fang[†‡✉],
Jiaxing Cheng[†‡], Dingyi Shi[†‡], Limin Sun[†‡]
[†]Institute of Information Engineering, CAS, China
[‡]School of Cyber Security, University of Chinese Academy of Sciences, China
{wangyuncheng, zhengyaowen, fangdongliang, chengjiaxing, shidingyi, sunlimin}@iie.ac.cn
[§]Tsinghua University, China; [¶]Ant Group, China
liupuzhuo.lpz@antgroup.com

***Abstract*—Robotic vehicles (RVs) play an increasingly vital role in modern society, with widespread applications in both commercial and military contexts. RV control software is the core of RV systems, which maintains proper operation by continuously computing the vehicle's internal state, sensor readings, and external inputs to adjust the system's behavior accordingly. However, the vast combination space of configurable parameters, command inputs, and environment-sensed data in RV software introduces significant security risks to the system. Existing fuzzing techniques face substantial challenges in effectively exploring this vast input space while uncovering deep bugs. To address these challenges, we propose ADGFUZZ, a novel fuzzing framework specifically designed to detect assignment statement bugs in RV control software. ADGFUZZ statically constructs an Assignment Dependency Graph (ADG) to capture inter-variable dependencies within the program. These dependencies are then propagated to the RV input space by leveraging naming similarities, resulting in a targeted set of inputs referred to as the matched input set (MIS). Building upon this, ADGFUZZ performs entropy-aware fuzzing over the MISs, thereby enhancing the overall efficiency of bug discovery. In our evaluation, ADGFUZZ uncovered 87 unique bugs across three RV types, 78 of which were previously unknown. All found bugs were responsibly disclosed to the developers, and 16 have been confirmed for fixing.**

I. INTRODUCTION

Robotic Vehicles (RVs) are widely deployed cyber-physical systems used in various applications, including commercial aerial photography [1], formation performances [2], [3], and military reconnaissance [4]. The operation of RVs, including flight attitude stabilization, real-time control, and autonomous navigation, is governed by control software within RVs. Despite their widespread use, RV systems have been found to contain various software bugs, which may lead to serious consequences such as unstable flight posture, loss of control or communication failure, mission failure, deviation from planned waypoints, or even catastrophic crashes. This highlights the necessity of automated techniques for systematically detecting such bugs.

✉: Corresponding Author
*: Both authors contributed equally to this work

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.231014
www.ndss-symposium.org

Several approaches have been proposed to detect bugs in RV control software [5], [6], [7], [8]. Given that RVs are exposed to a wide range of inputs, including system configuration parameters, control commands, and environmental factors such as wind and temperature, the testing input space is vast. To address the challenge, existing works primarily adopt two strategies to ensure testing efficiency. The first, as used in RVFuzzer [5], reduces redundant test inputs through techniques such as binary search and strategy composition. The second, as seen in PGFuzz [8], attempts to discover bugs using a limited set of rule-based strategies extracted from developer documentation. However, both approaches have notable limitations. In the first case, there is no control over which parts of the code are actually exercised during testing, making it unclear whether code regions with potential bug patterns are sufficiently explored. In the second case, a significant number of bugs in control computation formulas are missed, particularly those that are not documented.

Key Insight. Existing works reveal a fundamental limitation: the lack of prior knowledge about buggy code patterns or a strong reliance on documentation-driven analysis. As a result, the testing process may suffer from insufficient exploration of potentially buggy code regions. To address this limitation, we conducted an in-depth analysis of historical bugs and identified their associations with specific code patterns. Our investigation revealed that the source code of RV control software encodes various physical formulas and mathematical models (e.g., flight dynamics) into program logic, which frequently involves assignment computations. A significant portion of bugs are rooted within these assignment statements. This is because converting physical formulas and mathematical models into corresponding assignment statements in code is inherently challenging and often prone to incompleteness, which increases the likelihood of introducing bugs. We refer to this category of issues as **assignment statement bugs**. Based on this finding, we conducted an empirical study of RV software issue reports over the past decade. The analysis revealed that, among 207 semantic and logical defects, 28.02% were assignment statement bugs directly caused by incorrect parameter assignments. The remaining 71.98% were due to missing inputs and were mostly discovered and fixed during development, making them difficult to find through testing. Motivated by these findings, we aim to propose an automated fuzzing technique that can efficiently detect assignment statement bugs in RV systems.

Approach. To effectively detect bugs that may arise during assignment computation, we propose ADGFUZZ, an assignment dependency-guided fuzzing technique. Unlike previous works that blindly explore the vast input space of the RV, our approach focuses on assignment statements and fuzzes only the RV input parameters that influence these assignments within the RV implementation, thereby enabling more efficient and targeted testing. Our key observation is that assignment statements in RV control software involve numerous variables with inherent data dependencies during computation. These variables typically follow a consistent naming convention that contains rich semantic information, which often aligns with the naming patterns of the RV input parameters. Based on this insight, the high-level idea of our approach is to leverage naming similarities to identify RV input parameters associated with specific assignment statements and conduct targeted fuzz testing on them.

In the static analysis phase, we first propose **Assignment Dependency Graph (ADG)**, which captures interdependent assignment relationships among variables in the RV source code. Within each ADG, we identify **leaf variables (LVs)**, referring to variables that do not depend on any other assignments. We then apply fine-grained, term-based semantic matching to map each LV to the corresponding subsets of RV input parameters. We refer to these subsets as **matched input sets (MISs)**. As a result, the data dependencies between the specific assignment statement and the MIS are explicitly constructed. By generating test cases for MIS, ADGFUZZ focuses on a smaller, more error-prone segment of the input space, thus significantly improving the efficiency and effectiveness of the assignment statement bug detection.

During the fuzzing phase, given the large number of MISs, it is crucial to prioritize those with higher potential for discovering bugs. To this end, we compute multiple types of entropy for each MIS based on the ADG and perform entropy-aware MIS fuzzing, thereby improving the efficiency of bug discovery. Finally, due to the delay between injecting test inputs into the RV system and the eventual detection of anomalies, we accurately identify the test cases that truly triggered the observed anomalies and perform bug deduplication to eliminate redundant reports.

Evaluation. We evaluated ADGFUZZ on three widely used types of RVs. The results show that ADGFUZZ successfully identified 87 unique bugs, including 78 previously unknown ones. Among these, 29.88% could cause the vehicle to crash to the ground, 17.24% led to deviations from the designated trajectory, and 52.87% triggered memory overflows. All identified bugs have been responsibly reported to the developers. As of this writing, 16 bugs are under consideration for future patches. We further compare ADGFUZZ with the state-of-the-art policy-guided fuzzing framework PGFuzz and show that ADGFUZZ is capable of detecting 79 additional bugs that PGFuzz failed to uncover. To assess the effectiveness of the entropy-based prioritization strategy for MIS fuzzing, we conducted an ablation study. The results demonstrate that this strategy significantly accelerates the discovery of bugs and increases the number of bugs detected. Finally, we provide an in-depth analysis of three representative cases discovered by ADGFUZZ to illustrate how these bugs can be exploited by adversaries to compromise a running RV system.

Contributions. Our paper makes the following contributions.

- We proposed ADGFUZZ, a novel framework for the efficient detection of assignment statement bugs. The prototype of ADGFUZZ was implemented with approximately 3,000 lines of Python code.
- We proposed an entropy-aware MIS fuzzing technique, which prioritizes input subsets with higher potential for triggering bugs.
- We evaluated ADGFUZZ on three widely used types of RVs. The results show that ADGFUZZ outperforms state-of-the-art approaches in bug detection, identifying a total of 87 unique bugs, including 78 previously unknown.
- We released the source code of ADGFUZZ on GitHub[1] to facilitate reproducibility and further studies.

II. BACKGROUND

RV Control Software. RV control software is the core computational system that enables RVs to process information and make decisions. It handles motion control, environmental sensing, task planning, and real-time decisions. The software consists of three main modules: (1) sensor module: periodically collects real-time input signals from onboard sensors (e.g., LiDAR, IMU); (2) control logic module: processes sensor data, reference states, and task requirements. It applies motion planning and decision-making algorithms to generate appropriate control signals; and (3) communication module: maintains bidirectional communication with the ground control station (GCS). It receives user commands while transmitting vehicle status, task progress, and feedback, supporting remote monitoring and coordinated operation.

RV Input Space. RV inputs refer to the data that the RV can receive, mainly including three categories: (1) configuration parameters, (2) control commands (sent by the GCS or loaded from mission files), and (3) environmental data from onboard sensors. Before takeoff, the RV reads and initializes configuration parameters from files containing default values. Users can adjust these parameters before takeoff to preset the RV's operating state and thresholds, such as the maximum deflection angle and acceleration. These parameters can also be modified after takeoff. In the RV simulator, environmental data can be adjusted in the same way as sending commands. For example, ArduPilot [9] includes an SITL [10] simulator

[1]https://github.com/wyunc/ADGFuzz

that allows testing the RV control software's correctness in a simulated environment. By setting `SIM_WIND_SPD`, users can modify the wind speed in the simulation.

Due to the complexity of RV control software, the number of RV inputs may be over 1,000. For example, ArduCopter [11] includes more than 4,000 configuration parameters and simulation environment variables, along with 164 control commands [12]. Among them, over 200 parameters regulate RV attitude and rate control, while more than 500 are related to sensors. Different types of inputs can influence the same system variable. For example, the configuration parameter `MOT_THST_EXPO` adjusts the throttle curve, whereas the control command `MAV_CMD_DO_THROTTLE` directly sets the throttle percentage. Therefore, when an RV receives multiple inputs, these inputs may affect the same physical quantity. Conflicting effects can lead to inconsistent system states. Moreover, since inputs can directly modify critical data, even a single input may trigger abnormal behavior. Unfortunately, RV control software rarely considers handling such cases.

III. Motivation

In this section, we present an interesting example that serves as the motivation for this work (§III-A). We then introduce an exploratory analysis to demonstrate the prevalence of this issue in popular RV control software (§III-B). Finally, we describe the threat model considered in this paper (§III-C).

A. *Motivating Example*

In RVs, the virtual reference point represents an expected position generated by the trajectory planner to guide flight control and path planning. Since determining the actual position of an RV requires complex sensor fusion, the virtual reference point provides a computationally efficient and smoother basis for subsequent flight decisions. In the vehicle's control logic, task completion is typically judged by whether this virtual reference point has reached the designated waypoint within a predefined tolerance radius. However, discrepancies inevitably exist between the virtual reference point and the RV's true position. The virtual reference point continues to advance along the planned path regardless of wind drift or control errors, which may lead the system to mistakenly mark the task as complete even when the vehicle has significantly deviated from its intended trajectory.

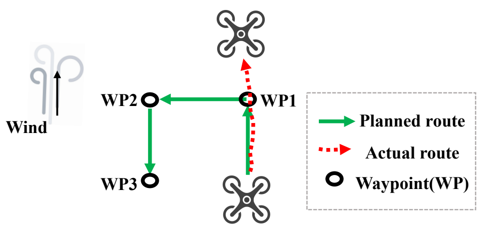


Fig. 1. Wind and misconfiguration caused the drone to deviate.

For example, we illustrate the code before and after fixing the virtual reference point calculation bug in Figure 2. The

```
bool AC_WPNav::advance_wp_target_along_track(float dt) {
    float track_error = _pos_control_error.dot(track_direction);
    float track_velocity = _inav_velocity_neu.dot(track_direction);
-   float track_scaler_dt = constrain_float(0.05f + (track_velocity - get_pos_kP() *
                                            track_error) / curr_target_vel.length(), 0.1f, 1.0f);
+   float track_scaler_dt = constrain_float(0.05f + (track_velocity - get_pos_kP() *
                                            track_error) / curr_target_vel.length(), 0.0f, 1.0f);
    float track_scaler_tc = 0.01f * get_wp_acceleration()/_wp_jerk;
    float _track_scalar_dt += (track_scaler_dt - _track_scalar_dt) * (dt / track_scaler_tc);
    bool s_finished =  _s_leg.advance_target_along_track(_wp_radius_cm, get_corner_accel(),
                       _flags.fast_waypoint, _track_scalar_dt * vel_scaler_dt * dt,
                       target_pos, target_vel, target_accel);
    if (!_flags.reached_destination) { // check whether reached the waypoint
        if (s_finished) {
            _flags.reached_destination = true;
  //...(omitted). This means the change in angle is equivalent to the change in acceleration.
```

Fig. 2. Before and after fixing the virtual reference point calculation bug in RV navigation code.

variable `s_finished` serves as a flag indicating whether the drone has reached the target waypoint. Its value depends on a series of variables, including `_track_scaler_dt`, which in turn is influenced by the parameter `track_scaler_dt` (buggy code). The `track_scaler_dt` parameter, which controls the advancement rate of the virtual reference point, was set to a minimum value of 0.1. This caused the system to continue advancing the virtual reference point at a minimum rate of 10%. Under normal conditions, this mechanism ensures that the virtual reference point advances at a reasonable speed and does not stall, which is generally effective. However, in the presence of strong winds, this calculation can become problematic. As the actual position of the copter is still hundreds of meters away from the waypoint due to wind interference (as shown in Fig. 1), the virtual reference point continues to accumulate progress and may eventually surpass the waypoint threshold (e.g., WP2), leading to incorrect mission completion. In other words, the incorrect computation of `track_scaler_dt` propagates through the assignment chain, causing the error to accumulate. Eventually, this affects `s_finished`, leading the drone to mistakenly determine that it has reached the waypoint.

However, existing bug detection approaches for RVs do not directly focus on assignment statement bugs. For example, tools like RVFuzzer [5] and LGDFuzzer [7] mainly target issues caused by improper settings of critical sensor input parameters such as position and velocity of x, y, z-axis and pitch, yaw, roll angles. PGFuzz [8] relies on manually crafted rules to define safety policies, for instance, whether a parachute can be deployed below a certain altitude, to identify safety violations.

B. *Exploratory Study*

To investigate whether bugs caused by assignment statements are prevalent in RVs, we conducted an exploratory study on issue reports from the open-source software ArduPilot. We analyzed all 819 issues labeled as "BUG" over the past 10 years (from January 2015 to January 2025). It is important to note that we filtered out issues unrelated to semantic or logical bugs (e.g., compilation errors, display failures, feature enhancements, and algorithm adjustments).

By examining the fixed code in these issues, we categorized them into five types: corrections to assignment statements, additions or deletions of logic, modifications to function calls,

Table I. Bug fix patterns in 'BUG'-labeled ArduPilot issues and their categorization.

Fixing code types	# of issues	Proportion (%)
Assignment statement corrections	58	28.02
Logic additions or deletions	111	53.62
Function call modifications	16	7.74
Function return type or numerical unit changes	11	5.31
Default value adjustments	11	5.31

changes to function return types or numerical units, and adjustments to default values. The results are shown in Table I. We found that 28.02% of the bugs were directly caused by incorrect parameter assignments. Based on this observation, we designed ADGFUZZ to detect bugs originating from such patterns. The remaining 71.98% of bugs were caused by missing inputs, and most were discovered and fixed during development, making them nearly impossible to reproduce.

C. Threat Model

In this paper, we focus primarily on implementation bugs in assignment statements. Since many configuration parameters and control commands in RV software are user-configurable, the presence of such bugs can make the system sensitive to misconfigurations. A user may unintentionally set values that exceed typical or reasonable boundaries, thereby triggering unintended behavior and compromising the safety of the RV. Therefore, our approach mainly facilitates software quality assurance (QA) by addressing implementation bugs in assignment statements.

In addition, there is a certain possibility that an attacker gains control of the RV software. Such attacks might include tactics like GCS spoofing [13] or communication link hijacking [14], [15]. Once control is obtained, the attacker may attempt to issue destructive commands, such as disabling throttle output, forcing the drone to land, or disarming it. Although these actions can be effective, they are usually easy to detect [16], [17] and can be blocked by mission monitoring mechanisms [18]. In contrast, bugs in assignment statement implementations may allow attackers to interfere with system behavior in a more concealed manner. This provides an opportunity for disruption that is significantly harder to detect using conventional monitoring tools. Therefore, detecting such bugs also helps eliminate potential attack paths that could be exploited by an adversary with access to RV configuration or input interfaces.

IV. APPROACH

A. Overview of ADGFUZZ

To effectively detect assignment statement bugs, we propose ADGFUZZ, an assignment-oriented fuzzing technique. Unlike prior works that blindly explore RV's vast input space, our approach focuses on assignment statements and fuzzes only the input parameters that influence these assignments within the RV implementation, thereby enabling more efficient testing. Figure 3 illustrates the overall workflow of ADGFUZZ. ADGFUZZ first extracts assignment statements and their corresponding assignment chains from each function in the source code to construct an assignment dependency graph (ADG), which represents the interdependence between variables (§IV-B). Next, we use leaf node variable names in the ADG to infer relevant inputs and group them into MISs (§IV-C). Subsequently, we define a customized entropy-based metric to evaluate each MIS, converting the entropy values into a probability distribution that assigns more energy to higher-entropy sets during fuzzing (§IV-D). Then, based on the probability distribution of MISs, we sample an MIS at each iteration to generate a test case, which is then executed in the RV simulator to identify potential bugs (§IV-E). Finally, during post-processing, we perform input minimization to refine the test cases. The goal is to identify the shortest input capable of triggering a bug while eliminating redundant inputs (§IV-F).

B. Assignment Dependency Graph Building

We introduce the concept of an *assignment dependency graph* to represent data-flow dependencies in assignment operations between dependent variables and independent variables within a function. The *assignment dependency graph* is derived from *assignment statements*, which are defined as follows:

Definition 1 (Assignment Statement)**.** *An* assignment statement *(AS) is a tuple* $\mathbb{S} = (y, \boldsymbol{X}, \boldsymbol{O})$ *with: (1)* $y \in V_{dep}$*, where* V_{dep} *is a set of dependent variables on the left-hand side (LHS) of an assignment expression, (2)* $\boldsymbol{X} = \{x_1, ..., x_n\} \subseteq (V_{ind} \cup \mathbb{F})$ *where* V_{ind} *is a set of independent variables (operands) on the right-hand side (RHS) of an expression statement,* $\mathbb{F}$ *denotes function invocations (e.g.,* $x.\text{func}()$, $\text{caller}(x)$*), and (3)* $\boldsymbol{O} \subseteq \{\oplus_1, ..., \oplus_m\}^{n-1}$*, where each* $\oplus_i$ *is an operator (arithmetic or logical) used in the expression.*

$$\oplus_i \in \underbrace{\{+, -, \times, \div\}}_{arithmetic} \cup \underbrace{\{\wedge, \vee, \neg, =\}}_{logical}$$

An *assignment dependency graph* (ADG) is constructed from one or more interdependent assignment statements within a function. Given the definition of *assignment statement*, *ADG* is defined as follows:

Definition 2 (Assignment Dependency Graph)**.** *An* assignment dependency graph *is a directed acyclic graph* $\mathbb{G} = (V, E)$ *with:*

- $V = V_{root} \cup V_{leaf} \cup V_{semi}$ *is a set of nodes where:*

$$V_{root} = \{v \mid \exists \mathbb{S}_1 : v \in \mathbb{S}_1.y \wedge \nexists \mathbb{S}_2 : v \in \mathbb{S}_2.X\}$$
$$V_{leaf} = \{v \mid \exists \mathbb{S}_1 : v \in \mathbb{S}_1.X \wedge \nexists \mathbb{S}_2 : v \in \mathbb{S}_2.y\}$$
$$V_{semi} = \{v \mid \exists \mathbb{S}_1 \neq \mathbb{S}_2 : v \in \mathbb{S}_1.y \wedge v \in \mathbb{S}_2.X\}$$

- $E = E_{RL} \cup E_{RS} \cup E_{SL} \cup E_{SS}$

where V_{root} *contains only a single root node,* V_{leaf} *is a set of nodes that exclusively receive incoming edges,* V_{semi} *refers to semi-dependent nodes, which act as independent variables in some assignment statements and as dependent variables in others.* E_{RL} *is a set of directed edges pointing from a node in* V_{root} *to a node in* V_{leaf}*. Similarly,* E_{RS} *represents directed*

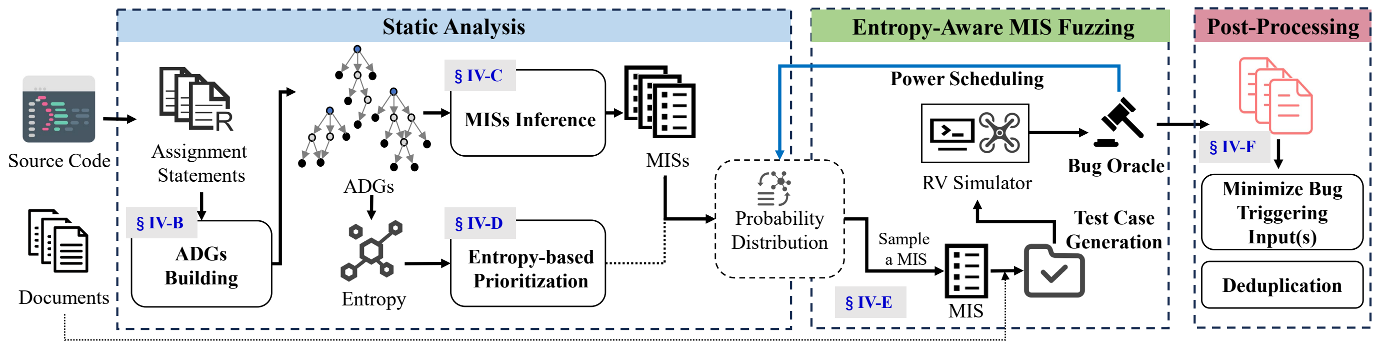


Fig. 3. Overview of ADGFUZZ.

edges from nodes in V_{root} to those in V_{semi}, E_{SL} represents directed edges from V_{semi} to V_{leaf}, and E_{SS} consists of directed edges connecting nodes within V_{semi}.

Throughout the rest of this paper, for simplicity, we use n_{R}, n_{L}, and n_{S} to denote nodes in V_{root}, V_{leaf}, and V_{semi}, respectively.
ADG Building. ADGFUZZ constructs assignment dependency graphs from the source code, incorporating forward syntax filtering and backward assignment analysis. First, we extract only the lines of code within functions that satisfy the definition of an assignment statement. And we filter and modify the variable names by removing numerical characters and replacing function calls; for example, converting `func(var1, var2_21)` into `func_var1` and `func_var2`. Next, we encode the properties of each variable node based on the assignment relationships. Finally, we establish edge connections by integrating both the explicit dependencies between variables and the implicit dependencies introduced by semi-dependent bridging nodes.

Algorithm 1 describes in detail the steps for ADGFUZZ to construct the ADG. ① It processes the source code of a single function at a time and performs forward syntactic filtering based on pattern matching, retaining only assignment statements by removing all other lines of code (Line 3-4). ② It takes the identified S as input, sorts them according to their order of appearance (Line 5). For each statement, the variable on the LHS is added to Set_{dep} as a node, with its property (p) initialized to `null` (Line 10). ③ It starts from the last node in Set_{dep} and performs backward traversal to find the first node with a `null` property, denoted as n (Line 13). The index of its corresponding assignment statement is i (Line 25). It marks the property of n in Set_{dep} as `root` and adds it into V_{root} (Line 14-15). ④ It finds all statements in S whose index is less than i and which use n as their LHS. The RHS variables in these statements are added to Set_{ind} (Line 26-28). ⑤ It constructs edges from the current node n to each node in Set_{ind} (Line 31). ⑥ Then, for nodes that appear in both Set_{dep} and Set_{ind}: it sets the property of these nodes in Set_{dep} as `semi`, inserts them into Set_{temp}, and deletes them from Set_{ind}. The remaining nodes in Set_{ind} are added to V_{leaf} as leaf nodes (Line 32-37). ⑦ For each node in Set_{temp}, it removes one node n at a time and inserts it into V_{semi} (Line 37). The index of the assignment statement corresponding to node n is i. Repeat steps ④-⑦ until Set_{temp} is empty. ⑧ Finally, it deletes nodes in V_{leaf} that are also present in V_{root} to eliminate potential loops, in coordination with step ④ (Line 18). ⑨ This process yields one complete assignment dependency graph starting from a root node. The algorithm repeats the above steps until all nodes in Set_{dep} have been assigned a non-null property, thereby generating the complete set of assignment dependency graphs for the function.

Once the ADGs are constructed, we assume that every pair of non-root nodes in each graph is assignment-dependent, and that the variables represented by the leaf nodes maintain the simplest form of assignment dependency. We use the leaf nodes of an ADG as indicators to infer matched input sets (MISs) with potential interdependent relationships (§IV-C). Furthermore, each MIS is ranked by the entropy of its associated ADG, enabling execution prioritization (§IV-D).

C. MIS Inference

ADGFUZZ infers a test input set from each ADG that may influence the value of the root node variable. Since program variables and input parameters jointly influence the RV control software through internal logic execution and real-world physical behavior, their correlation offers an opportunity for input inference. In addition, they often follow consistent naming conventions [19], [20]; for example, both may include the same sensor abbreviations. Therefore, we adopt a term similarity association strategy to infer MISs from LVs, leveraging the consistent naming conventions shared between program variables and input parameters.

Figure 4 illustrates how an MIS is generated from an ADG. First, we extract LV names from ADG leaf nodes and split them into variable terms using underscores as delimiters. Prior to this, the names of all RV-supported multi-source inputs are also segmented into input terms. Second, we apply a predefined term association table to replace and extend certain terms, while filtering out verbs (e.g., "get") and terms shorter than two characters. Finally, we match the transformed variable terms with the RV input terms, include the inputs corresponding to matched terms in the MIS, and record the number of matched terms associated with each input. A representative ADG-MIS example is provided in §IX-B.

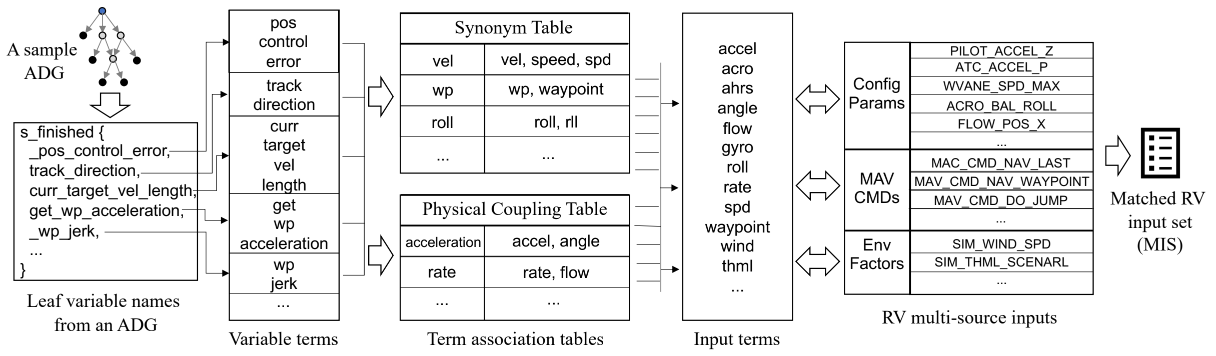


Fig. 4. An illustration of how MIS is inferred for a sample ADG based on term association.

Algorithm 1 Assignment Dependency Graph Construction

```
Input: Source code of a function $F$
Output: Assignment dependency graphs $G_s$ extracted from the function $F$
function ExtractS($F$)
    $S,\ G_s \leftarrow \emptyset$
    $AS \leftarrow$ ExtractAssignmentStatements($F$)        ▷ Retrieve AS from $F$
    $S \leftarrow$ FilterVariables($AS$)        ▷ Filter and normalize variable names
    $S \leftarrow$ SortByIndex($S$)        ▷ Initialize $S$'s index number: $S.index$
    $G_s \leftarrow$ BuildADG($S$)        ▷ Obtain all ADGs contained in $F$
    return $G_s$
end function
function BuildADG($S$)
    $Set_{\text{dep}} \leftarrow \{(\text{LHS}(s), \text{null}) \mid s \in S\}$        ▷ Initialize $Set_{\text{dep}}$
    $V_{\text{root}}, V_{\text{semi}}, V_{\text{leaf}}, E \leftarrow \emptyset$
    while $\exists(n, p) \in Set_{\text{dep}}$ with $p$ = null do        ▷ Iterate over $Set_{\text{dep}}$
        $(n, p) \leftarrow$ GetNullNode($Set_{\text{dep}}$) ▷ Get a node with $p$=null in reverse order
        $Set_{\text{dep}}$.UpdateP($n, root$)        ▷ Update the $n.p$ in $Set_{\text{dep}}$ to root
        $V_{\text{root}} \leftarrow \{n\}$
        $V_s, V_l, E_n$ = ProcessSemiNode($n, S$)
        $V_{\text{semi}} \leftarrow V_{\text{semi}} \cup V_s$
        $V_{\text{leaf}} \leftarrow (V_{\text{leaf}} \cup V_l) \setminus V_{\text{root}}$
        $E \leftarrow E \cup E_n$        ▷ Construct an ADG with $n$ as the root node
        $G_s \leftarrow G_s \cup (V_{\text{root}} \cup V_{\text{semi}} \cup V_{\text{leaf}}, E)$
    end while
    return $G_s$
end function
function ProcessSemiNode($n, S$)
    $i \leftarrow$ GetStaIndex($n$)        ▷ Retrieve the index of the $AS$ corresponding to $n$
    for $s \in S$ do        ▷ Get nodes used on the RHS of the $S$ where $n$ is the LHS
        if $s.index < i$ and $n \in$ LHS($s$) then
            $Set_{\text{ind}} \leftarrow$ RHS($s$)
        end if
    end for
    $E \leftarrow \{(n, v) \mid v \in Set_{\text{ind}}\}$        ▷ Build edges from node n to nodes in $Set_{\text{ind}}$
    $Set_{\text{temp}} \leftarrow Set_{\text{ind}} \cap Set_{\text{dep}}$        ▷ Get semi-dependent nodes
    $Set_{\text{ind}} \leftarrow Set_{\text{ind}} \setminus Set_{\text{temp}}$        ▷ Preserve leaf variables
    $V_{\text{leaf}} \leftarrow V_{\text{leaf}} \cup Set_{\text{ind}}$
    for $v \in Set_{\text{temp}}$ do        ▷ Update the property of semi-nodes in $Set_{\text{dep}}$
        $Set_{\text{dep}}$.UpdateP($v, semi$)
        $V_{\text{semi}} \leftarrow V_{\text{semi}} \cup \{v\}$
    end for
    return $V_{\text{semi}},\ V_{\text{leaf}},\ E$
end function
```

We define two types of term association tables. (1) Synonym table: Includes synonyms and abbreviations. Some terms may appear in different lexical forms depending on usage, such as `velocity` and `speed`, or `location` and `position`, which are treated as equivalent. In addition, abbreviations are commonly used in code. These are typically formed by retaining only the initial letters of a long word (e.g., `velocity` becomes `vel`) or by omitting vowels from the end (e.g., `roll` becomes `rll`). (2) Physical coupling table: This table captures coupled relationships between physical concepts. For instance, in a multi-rotor vehicle, the thrust direction is determined by the vehicle's attitude, particularly its tilt angle. When the vehicle tilts, the horizontal component of the thrust induces acceleration. Given thrust T, tilt angle θ, and RV mass m, the horizontal force is $F_x = T \cdot \sin\theta$, and the resulting acceleration is $a = F_x/m = T \cdot sin\theta/m$, which approximates to $T \cdot \theta/m$. This means that variations in angle directly lead to variations in acceleration. Therefore, angle and acceleration are treated as an associated term pair in this table.

The synonym table was manually constructed based on domain expert knowledge and previous work [8], [21]. In contrast, the physical coupling table is semi-automatically derived from source code comments, which typically follow established readability guidelines [22] to explain the purpose of functions, methods, and specific computational statements, often describing relevant physical quantities. For example, Line 11 in Figure 2 describes the equivalence relationship between angle and acceleration changes. We extract all comments from the RV codebase and employ a large language model to identify physical quantity terms that are potentially coupled, thereby forming the physical coupling table. This process is carried out as a one-time effort. Furthermore, we observed that most control-related variables and physical relationships are conceptually consistent across different RV modules (e.g., Copter and Plane), allowing these tables to generalize well. Additionally, when deployed across platforms (§VIII-C), the term association tables constructed for ArduPilot successfully supported MIS construction for PX4.

D. Entropy-based Prioritization

Each ADG consistently yields one MIS, which represents a heuristic and non-deterministic group of semantically related inputs. While not guaranteed to be precise, these inputs are often correlated, and some are likely to jointly drive the RV into deeper and potentially more error-prone state spaces. To enhance fuzzing efficiency and increase the likelihood of uncovering diverse bugs, we prioritize MISs with a higher potential to trigger unexpected behaviors. Traditional methods, such as random selection or coverage-guided fuzzing, tend to focus on the execution behavior of individual inputs and struggle to quantify the latent cooperative potential among inputs within a MIS. According to Shannon's information theory [23], entropy quantifies the uncertainty of information, where higher entropy indicates greater unpredictability and richer information content. Therefore, we define an entropy metric for each MIS to capture its likelihood of triggering unknown state transitions. During fuzzing, we allocate more resources to MISs with higher entropy values.

Specifically, we quantify the entropy of each MIS based along two dimensions: (1) the number of semi-dependent nodes in its ADG, and (2) the semantic relevance of its leaf nodes, measured by how many of their valid terms align with RV input terms. We denote the entropy contributions of these two dimensions by E_{num} and E_{qual}.

Given that RV control software typically runs on embedded platforms (e.g., APM), developers are unlikely to introduce a large number of redundant alias assignments that consume unnecessary memory. Therefore, an ADG with more semi nodes suggests richer intermediate computations, indicating a higher degree of latent semantic information. For an ADG with N semi nodes, the entropy of its MIS is defined as:

$$E_{\text{num}}(\text{MIS}) = \log_2(|N| + 1) \tag{1}$$

For each LV within the ADG, its terms are considered relevant only if they match with the input terms of the RV. To reflect this semantic alignment, we define a quality-based entropy metric that quantifies the number of valid terms contained in each LV. The quality-based entropy for the MIS generated from an ADG is calculated as follows:

$$E_{\text{qual}}(\text{MIS}) = \log_2\left(1 + \sum_{v \in V_{\text{leaf}}} M(v)\right) \tag{2}$$

where $M(v)$ denotes the effective information entropy of a variable v. A variable with more matched terms is considered to carry more semantic information. However, if a term corresponds to a larger number of RV inputs, it is considered less informative. Therefore, the effective entropy of v is positively correlated with the number of its matched terms and negatively correlated with how commonly each term appears among the inputs. Let v contain k terms. When using the i terms in v, suppose they collectively match T_i unique RV inputs. Then, the effective information entropy $M(v)$ of v is defined as:

$$M(v) = \sum_{i=1}^{k} \frac{i}{T_i} \tag{3}$$

The total entropy score of a MIS is given by:

$$E(\text{MIS}) = \log_2(|N| + 1) + \log_2\left(1 + \sum_{v \in V_{\text{leaf}}} \sum_{i=1}^{k} \frac{i}{T_i}\right) \tag{4}$$

Each MIS is assigned an entropy value as defined by Equation (4). During fuzzing, MISs are selected for test case generation according to probabilities proportional to their entropy values. The probability of selecting a specific MIS is:

$$p(\text{MIS}) = \frac{E(\text{MIS})}{\sum_{L \in \text{MISs}} E(L)} \tag{5}$$

E. Entropy-Aware MIS Fuzzing

After obtaining the MIS using the procedure in §IV-C, we fuzz them according to the entropy metrics defined in §IV-D, enabling more efficient detection of assignment-related bugs. Algorithm IV-E outlines the fuzzing steps. It performs the following steps iteratively: ① Initialize a probability distribution P based on the entropy values of the MISs (line 2). ② Sample a set L from P (line 4). ③ (Re)initialize the RV simulator (line 5). ④ Based on the entropy value of L, determine the number of executions κ for this round, randomly select a subset of inputs from L, assign a value to each input, and execute them in the RV simulator (lines 7-12). ⑤ Detect anomalies in the RV using the bug oracle (line 13). ⑥ On anomaly, forward the executed test cases to post-processing (lines 14–15). ⑦ Dynamically adjust the entropy values of the tested L and the probability distribution P; remove MISs with entropy smaller than "1" (lines 16 and 19). ADGFUZZ repeats these steps until either the time limit τ set by the user is reached or the MISs are exhausted.

Algorithm 2 Entropy-Aware MIS Fuzzing

```
Input: RV Simulator SIM, inferred MISs $S_L$ from previous step, fuzzing time limit $\tau$
Output: A set of test cases $\text{Bug}_{\text{cases}}$ that can trigger bugs
 1: $\text{Bug}_{\text{cases}} \leftarrow \emptyset$                         ▷ Initialize the $\text{Bug}_{\text{cases}}$
 2: $P \leftarrow \text{Normalize}(\{L.\text{entropy} \mid L \in S_L\})$      ▷ Initialize the prob. dist.
 3: while $S_L \neq \emptyset$ & $\neg Timeout(\tau)$ do                 ▷ Main loop
 4:     $L \leftarrow \text{Sample}(S_L, P)$         ▷ Sample one $L$ from $S_L$ according to $P$
 5:     SIM ← SIM.reinitialize                      ▷ Reinitialize the simulator
 6:     $\text{Run}_{\text{cases}} \leftarrow \emptyset$
 7:     $\kappa \leftarrow$ CONVERT($L$.entropy)          ▷ Convert to execution times
 8:     while $\kappa > 0$ do
 9:         Inputs ← RANDOM($L$)                 ▷ Pick random inputs from $L$
10:         Cases ← ASSIGNVALUES(Inputs)  ▷ Assign value for each input
11:         $\text{Run}_{\text{cases}} \leftarrow \text{Run}_{\text{cases}} \cup$ Cases        ▷ Record executed test cases
12:         Msg ← SIM(Cases)                ▷ Record RV's status and logs
13:         Status ← ORACLE(Msg)       ▷ Determine whether a bug occurs
14:         if Status.$abnormal()$ then
15:             $\text{Bug}_{\text{cases}} \leftarrow \text{Bug}_{\text{cases}} \cup \text{Run}_{\text{cases}}$      ▷ Record execution cases
16:             UPDATEBUGENTROPY($L$)                ▷ Update $L$'s entropy
17:             $\kappa \leftarrow 0$
18:         end if
19:         UPDATEENTROPY($L$)                       ▷ Update $L$'s entropy
20:         $\kappa \leftarrow \kappa - 1$
21:     end while
22: end while
23: function UPDATEENTROPY($L$)
24:     $L$.entropy ← $L$.entropy/2                ▷ Adjust the entropy of $L$
25:     $P \leftarrow \text{Normalize}(\{L.\text{entropy} \mid L \in S_L\})$              ▷ Reinitialize $P$
26: end function
```

Fuzzing Phase Workflow. The fuzzing process consists of three interacting modules running in parallel: (1) a Simulation Module, (2) an Execution Module, and (3) an Oracle Module. The Simulation Module launches a new SITL instance at the beginning of each iteration, loads a multi-phase navigation mission, and executes it automatically. The Execution Module first samples a MIS from an entropy-based probability distribution as the input injection template for the current round, and sends the test input to the SITL once the RV enters the armed state (including takeoff, navigation, turning, etc.). Meanwhile, the Oracle Module runs in parallel during input execution to detect anomalies in the RV system. When a bug is detected or the execution times out, the entropy of the current MIS is updated and the probability distribution is adjusted in real time. The triggered test sequence is then passed to the post-processing module for further analysis.

Unlike traditional fuzzers that mutate raw bytes or isolated parameters, ADGFUZZ performs input injection within a high-fidelity simulation environment (SITL) under full mission execution, including waypoint uploading, command transmission, parameter adjustment, and environment simulation. This design ensures that the fuzzing process closely reflects real RV operational workflows and exposes errors that may arise under realistic mission conditions.

Power Scheduling for MISs. During the fuzzing process, a high-entropy MIS may produce numerous test cases, potentially containing diverse input combinations that expose different bugs. If a traditional fail-fast strategy is used, where testing stops after detecting the first bug, additional bugs within the same MIS may remain undetected. Conversely, excessive testing of a single MIS can result in insufficient coverage of other MISs, thereby reducing overall testing efficiency. To strike a balance between in-depth bug discovery within individual MISs and broad coverage across multiple MISs, we propose a strategy that dynamically adjusts the entropy values and probability distribution of MISs.

We map the entropy value of each MIS to a sampling probability, prioritizing MISs with higher entropy for testing. After an MIS is selected, its entropy is reduced to lower the probability of being selected in subsequent rounds. Specifically, if the MIS triggers a bug, its entropy is halved, allowing for limited further exploration of the MIS that has already revealed a bug. If no bug is triggered within a round, its entropy is gradually reduced to lower the probability of continued exploration, until the allocated testing resources are exhausted, which promotes the coverage of other untested MISs. This strategy does not introduce biased exploration (see §IX-C), as it ensures both efficiency (high-entropy MISs with higher selection probabilities are more likely to discover bugs) and preserves the exploration of corner cases (low-entropy MISs will still be selected over time).

Test Cases Generation. In each round, we sample one MIS from the probability distribution as a template. A subset of its inputs is then selected to generate test cases. The number of executions for this MIS in the current round is determined by its entropy, since higher entropy indicates a greater number of input variables and thus requires more executions to sufficiently explore the combination space. Specifically, we map the unnormalized entropy value directly to the number of executions. Based on empirical observations, we enforce a minimum of 50 executions and cap the maximum at 500.

To explore different input combinations within an MIS, we randomly select a subset of its inputs for testing in each case. For each selected input, a value is assigned according to the following criteria: (1) If a valid range is defined in the official parameter or command documentation [12], [24], a value is sampled from within that range. (2) If no range is provided but a unit (e.g., rad, km) is specified, we define an appropriate value range for each unit and assign a value accordingly. (3) Otherwise, we extract the maximum ($\mathrm{In}_{\mathrm{max}}$) and minimum ($\mathrm{In}_{\mathrm{min}}$) values stated in the documentation, partition the interval [$\mathrm{In}_{\mathrm{min}}$, $\mathrm{In}_{\mathrm{max}}$] using a geometric progression, and assign a random value from this interval to ensure balanced coverage across different magnitudes. For example, if the range is [0, 100], ADGFUZZ generates values from [0,1), [1,10), and [10,100] with equal probability.

Bug Oracle. To enable automated detection of anomalies in RVs, we define three bug oracles, each targeting a specific invariant: (1) crash on the ground, (2) deviation from the planned route, and (3) software crash. These invariants address the most critical aspects of RV safety: whether the vehicle incurs physical damage, whether it can complete its intended mission, and whether its control software operates reliably. The design of these oracles is informed by the failure patterns observed in real-world bug reports analyzed in §III-B, empirical observations from simulation-based testing, and invariants used in previous RV-related bug detection studies.

First, crash on the ground refers to an airborne vehicle unexpectedly falling to the ground due to power failure or other faults, potentially causing irreversible hardware damage. We identify such bugs by monitoring system status messages and runtime logs generated by the vehicle. Second, the route deviation oracle monitors whether a vehicle performing waypoint navigation continuously drifts away from the target. If the distance to the destination increases over a short time window (e.g., 7 seconds), we classify the behavior as a deviation. Lastly, a software crash refers to memory-related errors, such as arithmetic or floating-point overflows, caused by improper parameter handling or missing safety checks in the control software. We consider a lack of heartbeat messages for two consecutive seconds as an indication of a software crash.

Handling Dynamic Runtime RV States. ADGFUZZ does not rely on explicit runtime state detection or reactive mechanisms. Instead, it proactively constructs a set of MISs during static analysis that are likely to trigger state transitions. These inputs are semantically correlated and can collectively interact with internal system logic to drive the RV into various deeper runtime states. In addition, during simulation, we load a multi-phase mission that spans multiple typical states (e.g., idle, takeoff, navigation), ensuring that fuzzing is conducted within the context of a complete mission. This design naturally triggers diverse runtime states during execution.

F. Post-Processing

Due to the unquantifiable delay between injecting test inputs into the RV system and the eventual manifestation of anomalies (see §VIII-A for details), we record the entire execution sequence from the selection of a given MIS to the detection of a bug. The post-processing stage aims to accurately identify the specific test cases responsible for triggering the observed anomalies. It receives a set of test cases $C = (C_1, \ldots, C_n)$ from the Bug Oracle, each linked to an observed anomaly. Then, it performs input minimization to derive a refined set $M = (M_1, \ldots, M_n)$, eliminating redundant inputs while preserving the ability to trigger the bug. Each test case set C_i contains all inputs executed by an MIS from the start of fuzzing until the bug was triggered, including inputs that are irrelevant to the bug. Minimizing these sets facilitates more efficient root cause analysis.

Bug-Triggering Input Minimization. To identify the minimal set of inputs that trigger a bug, we follow these steps: (1) For each test case set C_i, initialize `index` = 0 and launch a new RV simulator. The execution direction is encoded as 0 for forward and 1 for backward, with the initial direction set to forward (`order` = 0). (2) Execute test cases sequentially from the current `index` in the given `order`. After each execution, we introduce a delay τ to allow the effects of the input to fully propagate, where τ is empirically determined based on the bug type. (3) If executing a test case results in a bug, we update the `index` to the current position and include this test case in M_i. (4) Reinitialize the simulator to execute all test cases in M_i. If the bug reoccurs, M_i is considered the minimal triggering set. (5) Otherwise, we reverse the value of the execution direction `order` and repeat steps (2) to (4). In addition, when the time between bug triggering and manifestation exceeds a predefined threshold, the corresponding input set may become non-minimizable. In such cases, we apply a fallback strategy: remove inputs one by one and test whether the bug still occurs. Inputs that do not affect the occurrence of the bug are thus eliminated, allowing us to approximate a minimal set through this slower, iterative reduction process.

Bug Deduplication. Since different ADGs may have overlapping nodes and edges, the test case sets generated from different MISs may trigger the same bug. To address this, we deduplicate test case sets. Specifically, we group the minimized input sets by bug type. For any two minimized sets M_i and M_j of the same type where $i < j$, if they are identical, we retain the lower-indexed set M_i.

V. IMPLEMENTATION

We implemented ADGFUZZ in approximately 3,000 lines of Python code. Some key implementation details of individual modules are outlined below.

ADG Construction. We selected the source code from ArduCopter, ArduPlane, Rover, and the libraries module within the ArduPilot project [25]. Pattern-based string matching was employed to identify assignment statements and perform variable name normalization, which served as the basis for constructing the Assignment Dependency Graph (ADG).

MIS Generation and Entropy-based fuzzing. We extracted input names accepted by the RV control software from parameter [12], [26], [27] and command [24] documentation to build the input term set. Using the term association tables, we mapped ADG variable terms to RV inputs to construct MISs. Following the definition of information entropy presented in this paper, we computed and recorded the entropy value associated with each MIS. During fuzzing, we implemented the interaction with RVs based on the pymavlink library [28].

VI. EVALUATION

We aim to answer the following research questions.

RQ1: How effective is ADGFUZZ at detecting bugs?

RQ2: To what extent does entropy-based prioritization contribute to bug detection?

RQ3: How accurate and effective is the ADG–MIS construction pipeline in capturing the semantic intent of source code variables?

RQ4: How can the bugs discovered by ADGFUZZ be leveraged to affect the security of the RV?

A. Evaluation Settings

We selected three of the most widely used [29] RV types for our evaluation: quadcopter (Copter) [11], fixed-wing aircraft (Plane) [30], and unmanned ground vehicle (Rover) [31]. These models are drawn from ArduPilot [9], the most popular RV control software. We employed SITL [10] as the simulation environment for the RVs, MAVProxy [32] as the GCS to interface with the RVs, and MAVLink [33] as the communication protocol between the control software and the GCS.

All experiments were conducted on an Ubuntu 20.04 virtual machine equipped with an Intel Core i9-11950H processor, 32 GB of RAM, and Python 3.8.10. Each experiment was run continuously for 24 hours.

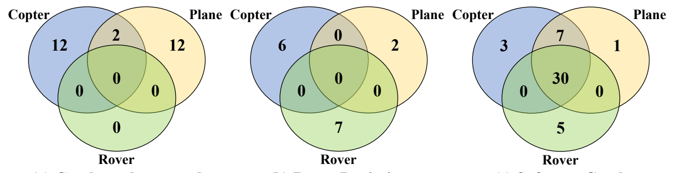
B. Finding Bugs (RQ1)

Table II summarizes the bugs discovered by ADGFUZZ across three types of vehicles. In total, ADGFUZZ identified 87 unique bugs, 78 of which were previously unknown. Because certain inputs (e.g., inertial sensor parameters) can be utilized across different vehicle types, some bugs were triggered by the same input in multiple vehicles. These cross-type bugs are counted once in our statistics. Figure 5 illustrates the distribution of discovered bugs across the three evaluated types of RVs. Specifically, we identified 14 bugs in both Copter and Plane that could lead to ground crashes; notably, two groups of inputs induced such failures in both vehicle types. As Rover is a ground-based vehicle, crash-to-ground bugs do not apply in its case. We also discovered 6, 2, and 7 route deviation bugs in Copter, Plane, and Rover, respectively, with no overlap observed across vehicle types. With respect to software crash bugs, 40 were detected in Copter, 38 in Plane, and 35 in Rover. Among these, 7 bugs were common to both Copter and Plane, while 30 were observed in all three vehicle types. Compared to Rover, more bugs are shared between the Copter and Plane. This is because these two types of RVs share a greater portion of control logic, such as takeoff and altitude control.

Analysis of Discovered Bugs. Among the 87 bugs (detailed in §IX-D) identified by ADGFUZZ, 26 (29.88%) bugs could cause airborne vehicles to crash to the ground, which may result in physical damage and potentially hit pedestrians if the crash occurs in a populated area. 15 (17.24%) bugs caused trajectory deviations, potentially leading to mission failure or

Table II. Bugs Discovered by ADGFUZZ Across Three Vehicle Types. *: When the same input triggers bugs in multiple types of RVs, we treat them as a single bug instance. These duplicates are excluded from our statistics to ensure accurate bug counting.

RV Type	# of bugs	Bug Invariants		
		Crash on the ground	Route deviation	Software crash
Copter	60	14	6	40
Plane	54	14	2	38
Rover	42	0	7	35
Total	87*	26*	15	46*



(a) Crash on the ground **(b) Route Deviation** **(c) Software Crash**

Fig. 5. The Venn diagram illustrates the overlap and distribution of the bugs we identified across different types of vehicles.

the false perception of mission completion, even when the vehicle had not reached the intended waypoint. 46 (52.87%) bugs triggered software crashes due to memory overflow.

Each bug was triggered by one or more inputs. Specifically, 77 bugs were caused by a single input, 11 were jointly triggered by two inputs, and one was the result of three inputs in combination. For instance, the parameter `AHRS_EKF_TYPE` has a valid range of 0, 2, 3, and 11; however, when set to 0, 2, or 11, it results in a floating-point overflow in the Rover. In another case, setting either `MOT_BAT_VOLT_MAX` = 45.42 or `MOT_PWM_MIN` = 8.3 individually within their valid ranges does not impact the operational stability of the Copter. However, when both are set simultaneously, the Copter experiences a sudden loss of thrust during flight and crashes to the ground (§VI-E1).

False Positives. We identified several inputs whose intended functionality inherently results in false positives. For instance, a drone must maintain a minimum motor output to achieve stable hovering or flight. Parameters such as `MOT_PWM_MAX` are explicitly designed to constrain motor output. When configured with overly low values, they can prevent the drone from generating sufficient thrust, resulting in a ground crash. Similarly, parameters like `SYSID_THISMAV`, which define the system identity or configuration of the RV, may induce operational disruptions when modified, such as breaking communication with the GCS. Detailed examples of such behavior are provided in §IX-A. These false positives are not included in the 87 bugs we report.

Comparison with Existing Work. PGFuzz [8] is a policy-guided fuzzing framework that emphasizes detecting violations of safety policies as documented. By comparing with its disclosed bug reports, we verified that 8 of the identified bugs were also detectable by PGFuzz. This indicates that there exists only a narrow intersection between the inputs inferred from source code in our approach and the policy-mapped inputs manually extracted by PGFuzz. Upon further analysis, we identified the following reasons why PGFuzz was unable to detect the remaining 79 bugs: when a bug arises from a simple concept without associated logical constraints, PGFuzz is unable to derive relevant detection policies, as its policy extraction mechanism is designed around strong logical rules such as "If ... then" or "must ...". Furthermore, not all implementation logic is documented. When developers introduce computational logic errors directly in the implementation, PGFuzz often fails to detect them, as the extracted policies are typically independent of such internal computations and therefore do not account for these types of bugs. For instance, as described in §III-A, the concept of a "vehicle reaching a waypoint" is intuitive, yet the implementation erroneously overlooks the vehicle's actual position, resulting in a bug. In contrast, ADGFUZZ begins with the waypoint arrival indicator variable as the root node and semantically traces all potential inputs that may affect this variable, enabling it to evaluate a broad spectrum of relevant inputs.

The scope of RVFuzzer [5] is limited to detecting control instabilities caused by configuration parameters, which prevents it from detecting the bugs identified by our approach. Therefore, we did not include RVFuzzer in our comparative analysis. Furthermore, we regard LGDFuzzer [7] as a functional subset of RVFuzzer, as it focuses exclusively on a very limited set of configuration parameters (20 in total), aiming to identify parameter values that may induce control instability in RV systems.

C. Effectiveness of Entropy-Based Input Prioritization (RQ2)

To evaluate the effectiveness of our entropy-guided strategy in enhancing RV bug detection, we developed a variant of ADGFUZZ named ADGFUZZ_NOE which substitutes entropy-guided input set scheduling with a random selection mechanism. We independently executed both ADGFUZZ and ADGFUZZ_NOE across three types of RVs, and the corresponding results are illustrated in Figure 6. Each data point in the figure denotes the timestamp at which a unique bug was first identified. Since individual bugs may be triggered multiple times, we record only the time of first occurrence. Lines labeled with "noe" indicate the number of bugs discovered by ADGFUZZ_NOE for each respective vehicle type.

As clearly demonstrated in Figure 6, when employing the complete strategy, ADGFUZZ exhibits a consistently high bug discovery rate within the initial hour of fuzzing. This indicates that the entropy-guided energy scheduling strategy effectively prioritizes input sets with a higher likelihood of triggering bugs. Relative to ADGFUZZ_NOE, ADGFUZZ discovered 18, 16, and 9 additional bugs in Copter, Plane, and Rover, respectively. Nonetheless, in the absence of entropy-based guidance, ADGFUZZ_NOE was still able to detect a substantial number of bugs within the first four hours. We attribute this outcome to two main factors. First, the RV control software often lacks parameter range validation, making overflow-related

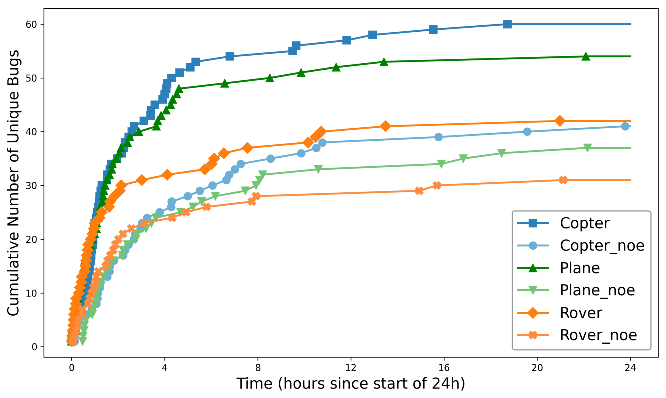


Fig. 6. Bug discovery trends over time for different strategies across three types of RVs.

Table III. Comparison of Bug Detection Performance in ADGFUZZ Variants

Subject	Unique Bug Detection			
	ADGFUZZ	ADGFUZZ-RANDOM	ADGFUZZ-NUM	ADGFUZZ-QUAL
Copter	60	34	47	55
Plane	54	31	42	49
Rover	42	25	33	38

bugs easier to trigger. Second, due to the inherent randomness of the selection process, ADGFUZZ_NOE occasionally samples high-entropy input sets by chance, some of which successfully trigger new bugs. This effect is observable in the intermittent bug discoveries that occurred between hours 4 and 12.

Contribution of Entropy Metrics. To evaluate the contribution of the entropy metrics (E_{num} and E_{qual} in §IV-E) to bug discovery, we performed ablation experiments on these two components. We implemented three variants of ADGFUZZ: (1) ADGFUZZ-RANDOM randomly selects inputs from the RV input space, (2) ADGFUZZ-NUM uses only E_{num} as the entropy metric for MISs, disabling E_{qual}, (3) ADGFUZZ-QUAL uses only E_{qual} as the entropy metric for MISs when calculating entropy. Table III shows the number of bugs discovered. Overall, ADGFUZZ-RANDOM exhibits the lowest bug discovery efficiency, as it cannot focus on high-risk areas and mainly detects software crashes. ADGFUZZ-NUM guides exploration of longer assignment chains by considering semi-dependent node counts, uncovering bugs from boundary conditions or overflows. However, it neglects more complex semantic information, such as the logical relationships between variables, leading to the inability to capture bugs that would only trigger in specific semantic contexts. ADGFUZZ-QUAL, by guiding the test to explore input spaces with higher semantic richness, is capable of identifying latent errors in code logic involving complex variable interactions, resulting in a higher bug discovery rate, as it better mimics the program's logical structure. By combining these two complementary strategies, ADGFUZZ enhances bug discovery.

D. Evaluation of ADG–MIS Construction (RQ3)

Accuracy. To evaluate the soundness of the ADG–MIS construction pipeline, we randomly sampled 150 ADG–MIS pairs generated during the static analysis phase. Two authors of this paper independently conducted manual cross-validation to assess the semantic correctness of the MIS inferred from each ADG. Specifically, we manually interpreted the semantic meaning of variable names in each ADG and identified terms containing essential contextual information, referred to as key terms. We then examined whether the corresponding MIS included RV inputs that could be reasonably inferred from these key terms, thereby verifying whether the MIS was semantically consistent with the ADG.

Among the 150 ADG–MIS pairs, 131 (87.33%) were deemed accurate. The remaining 19 semantically inconsistent cases fall into three categories: (1) Semantically unreliable: In 14 cases, the variable names carried no meaningful semantics and no key terms could be extracted. This occurred when the ADG was constructed from abstract or temporary variables, such as `Alpha`, `ptr_type`, or `default_list`. (2) Semantic ambiguity: In 3 cases, semantic ambiguity in the variable names caused the key terms to be missed during MIS construction. As a result, the MIS was not generated based on the correct key term, for instance with variables like `posvel`. (3) Semantic irrelevance: In 2 cases, key terms were identified but had no semantic correlation with RV inputs, typically representing external interfaces or application-level concepts, such as ROS `topics`. Notably, cases (1) and (3) do not lead to bug omissions since these variables lack actionable semantic information. As a result, the terms matched in these cases are typically generic (e.g., `type`, `default`), and the corresponding MISs tend to have low entropy values (as defined by our E_{qual}-based metric). Therefore, these MISs are unlikely to be selected during the fuzzing phase. In case (2), where semantic ambiguity exists, the confusing term fails to establish a semantic link with any RV input and is thus skipped during the matching process.

Effectiveness. We also investigated the frequency of naming matching failures and their potential impact on fuzzing performance. In ArduPilot, we identified 20,858 unique variables from the right-hand side of assignment statements. Our naming matching method successfully mapped 17,276 of these variables to RV inputs, while 3,582 (17.17%) failed to match. These mismatched variables could not establish a semantic connection with RV inputs and were thus skipped during the static analysis phase.

By collecting the MISs inferred from variables that successfully matched through naming, we found that they cover 99.98% (5005 out of 5006) of RV inputs. Therefore, the failure of naming matching does not lead to missed bugs or reduced coverage. This is because our name-matching method is robust to the small number of naming ambiguities. First, in most cases, MISs are not generated by a single variable but include multiple variables, making it less likely for all of them to fail matching. Additionally, the existence of non-ambiguous semantically equivalent contexts in other RV modules helps mitigate the impact of semantic matching failures. For example, the ambiguous term `posvel` appears only three times throughout the entire codebase, while the semantically meaningful subterms `pos` and `vel` co-occur in twelve different locations.

We note that when a semantically inaccurate MIS is sampled, it corresponds to testing a subset of the input space without the guidance of semantic information, which is unlikely to trigger bugs and leads to ineffective testing, thereby reducing fuzzing efficiency. However, our entropy-based prioritization mitigates this effect because these inaccurate MISs contain less useful information, have lower entropy, and are less likely to be selected during fuzzing. For instance, when we sampled 150 ADG-MIS pairs according to their entropy-based probability distribution, we found that 94.67% (142/150) of the inferences were semantically accurate.

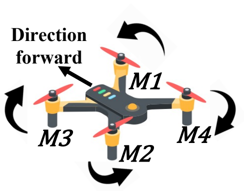
E. Effect of Exploitation (RQ4)

We present a detailed analysis of three representative examples identified by ADGFuzz to demonstrate how these bugs may be exploited by adversaries to compromise a running RV. Each case is accompanied by a root cause analysis to explain the underlying mechanisms leading to the observed failures.

1) Case Study 1 - Unexpected Voltage–Throttle Correlation Error Causing a Ground Crash: Each parameter remains within its documented valid range, and modifying either one individually does not trigger the bug. ADGFuzz identified that the simultaneous satisfaction of two specific conditions can cause the rotor motors to shut down, causing the vehicle to crash into the ground.

Attack. Consider a quadrotor hovering stably in mid-air (Figure 7a). An adversary can trigger a sudden loss of control and an uncontrollable crash by slightly increasing the maximum battery voltage parameter (`MOT_BAT_VOLT_MAX`) above the actual voltage level, while concurrently lowering `MOT_PWM_MIN`. Notably, adjusting either parameter individually does not affect the drone's stability. As battery voltage naturally decreases during discharge, configuring a higher maximum voltage prompts the firmware to continuously compensate under the assumption of full voltage, thereby sustaining stable thrust output. In addition, operators often interpret `MOT_PWM_MIN` as the minimum idle throttle; setting it to a lower value is intended merely to reduce idle propeller speed at 0% throttle and is not expected to cause failure.

Root Cause. As shown in Figure 8, the voltage compensation logic within the RV's flight control software adjusts motor thrust to offset battery voltage fluctuations. The compensation factor (`compensation_gain`) is approximately computed as the ratio of the configured maximum voltage to the actual voltage (Line 3). For example, if `MOT_BAT_VOLT_MAX` is set to $45.42V$ and the actual voltage is $42V$, then `compensation_gain` $\approx 45.42/42 \approx 1.08$. This leads to an approximate 8% amplification of all thrust-related control signals (thrust, roll, pitch, yaw) (Lines 4–8). The flight controller utilizes a mixer algorithm to combine these control signals and assign a normalized thrust coefficient (`_actuator[i]`) to each motor (Lines 9–10), which is subsequently mapped to a Pulse-Width Modulation (PWM) value (Line 14) and applied to the corresponding output channel to drive the respective motor (Line 15). However, for the rear-right motor (M4) on the quadcopter, the mixer coefficients for roll, pitch, and yaw are all -1 (Figure 7b). When amplified, these negative correction signals accumulate, resulting in $M4_{pwm} = thrust - roll - pitch - yaw$. With carefully crafted inputs, an attacker can cause `_actuator[3]` = 0. In such a case, M4's PWM output is solely governed by `pwm_min`, which is determined directly by the parameter `MOT_PWM_MIN`. Setting this parameter to an extremely low value causes the Electronic Speed Controller (ESC) to interpret the signal as lost, resulting in a total motor shutdown and an unrecoverable ground crash.



$$\begin{bmatrix} M1_{pwm} \\ M2_{pwm} \\ M3_{pwm} \\ M4_{pwm} \end{bmatrix} = \begin{bmatrix} 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 \\ 1 & -1 & -1 & -1 \end{bmatrix} \begin{bmatrix} thrust \\ roll \\ pitch \\ yaw \end{bmatrix}$$

(a) A quadcopter hovering in the air, powered by four propeller motors. (b) Mixer algorithm for adjusting motor thrust.

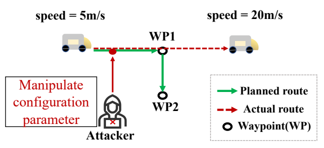
Fig. 7. (a) A quadcopter hovering in flight, stabilized by four propeller-driven motors. Arrows indicate the rotation direction of each propeller. (b) The mixer algorithm in the flight control software adjusts the thrust distribution across the four motors.

```
 1 void AP_MotorsMatrix::output_armed_stabilizing()
 2 { // apply voltage and air pressure compensation
 3     const float compensation_gain = thr_lin.get_compensation_gain();
 4     const float roll_thrust = (_roll_in + _roll_in_ff) ∗ compensation_gain;
 5     const float pitch_thrust = (_pitch_in + _pitch_in_ff) ∗ compensation_gain;
 6     float yaw_thrust = (_yaw_in + _yaw_in_ff) ∗ compensation_gain;
 7     float throttle_thrust = get_throttle() ∗ compensation_gain;
 8     float throttle_avg_max = _throttle_avg_max ∗ compensation_gain;

 9     req_i = throttle_thrust + roll_thrust ∗ mix_roll[i] +
               pitch_thrust ∗ mix_pitch[i] + yaw_thrust ∗ mix_yaw[i]; //Buggy code
10     _actuator[i] = constrain_float(req_i, 0.0f, 1.0f);

11 void AP_MotorsMulticopter::output_motor_mask()
12 {   const int16_t pwm_min = get_pwm_output_min();
13     const int16_t pwm_range = get_pwm_output_max() − pwm_min;
14     int16_t pwm_output = pwm_min + pwm_range ∗ _actuator[i];
15     rc_write(i, pwm_output);                                    //Buggy code
```

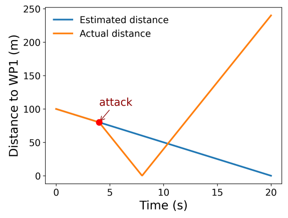
Fig. 8. A logic bug that results in the motor PWM output being set to an extremely low value.

2) Case Study 2 - Incorrect Waypoint Detection Due to Position Estimation Discrepancy: During navigation tasks, an RV does not always require physically reaching a waypoint to consider it "arrived." Instead, it precomputes a smooth trajectory between waypoints based on flight speed and inter-waypoint distance and estimates the time required to complete each segment. However, ADGFuzz revealed that relying on precomputed timing without verifying the RV's actual position can be exploited by adversaries to cause subtle route deviations, making the control software falsely assume waypoint completion even when the RV has deviated significantly.

Attack. Consider an autonomous delivery rover assigned to visit a sequence of waypoints for package distribution (Figure 9a). An adversary capable of modifying configuration parameters sets the `ATC_SPEED_FF` value far beyond its safe range. This manipulation of the speed feedforward gain causes the rover to accelerate significantly beyond its intended

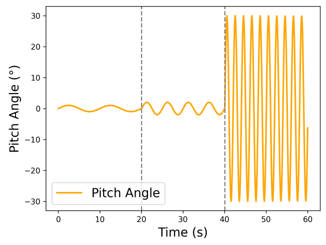


(a) The planned vs. actual trajectory of the ground vehicle.

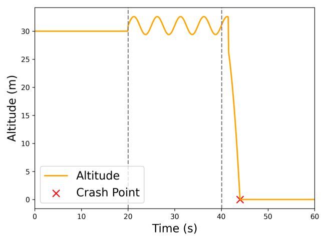


(b) Comparison between the RV-estimated and actual distances to the target waypoint.

Fig. 9. An attacker launches an attack at the 4th second after the vehicle starts its delivery task, causing a sudden surge in speed. However, the RV control software continues to use incorrect logic for calculating the navigation distance.



(a) Pitch angle over time.



(b) Altitude over time.

Fig. 10. Changes in pitch angle and altitude of the plane under attack.

velocity, reaching 20 m/s instead of the nominal $5m/s$, as depicted in Figure 9b. Despite the rover rapidly passing by the waypoints, the RV control software continues to rely on flawed logic when calculating the vehicle's distance to each waypoint. As a result, it erroneously concludes that the rover has successfully reached each navigation point, even though it has not. Consequently, all packages are incorrectly marked as delivered, leading to mission failure, lost cargo, and a heightened risk of collisions along the unintended route.

Root Cause. This bug stems from two primary factors. First, `ATC_SPEED_FF` is not constrained by effective safety bounds. When set too high, it leads the speed controller to apply disproportionately large throttle outputs, causing the vehicle to operate at speeds far exceeding expectations. Second, the fast waypoint transition mechanism relies primarily on an internal timer derived from the basic kinematic relationship (distance = speed × time) rather than direct verification of the vehicle's actual position. When the feedforward gain is increased excessively, the vehicle's actual speed diverges from the nominal speed assumed in the timer calculation, invalidating the timer as a proxy for position. Nevertheless, the RV control logic continues to rely on this flawed estimate, leading to misjudged waypoint arrivals and eventual task failure.

3) Case Study 3 - Plane Crash Caused by Incorrect Pitch Gain Configuration: ADGFuzz identified that a mechanism designed to smooth pitch dynamics, when misconfigured, can instead destabilize the aircraft and ultimately result in a crash.

Attack. Consider an aircraft engaged in a reconnaissance mission, operating along a predetermined low-altitude flight path to monitor the coastline or perform mapping tasks. An adversary with the capability to manipulate or overwrite configuration parameters may modify the parameter `PTCH_RATE_D_FF`, which governs the magnitude of pitch control, either prior to or during the flight. They may also constrain the permissible range of downward pitch by setting the minimum pitch angle to a positive value, thereby preventing the aircraft from achieving a nose-down attitude. Under such constraints, when the aircraft encounters atmospheric turbulence or requires altitude reduction, the inability to achieve a sufficient nose-down pitch results in abrupt and excessive pitch responses. This dynamic destabilizes the flight control system, ultimately leading to an uncontrolled descent and crash.

Root Cause. The identified bug stems from a conflict between the pitch control and throttle management safety logics in the RV control software. The Total Energy Control System (TECS) is designed to automatically reduce throttle when the aircraft's nose is pointed downward, thereby leveraging gravitational force to assist in descent. However, if the minimum pitch angle configuration parameter `TECS_PITCH_MIN` is set to a positive value, the condition `_PITCHminf` < 0 can never be fulfilled, effectively disabling the logic branch responsible for throttle reduction during nose-down flight. As a result, even when the nose is slightly pitched downward, the system is unable to reduce throttle and is instead compelled to maintain or increase engine thrust. Once the throttle becomes effectively locked, TECS attempts to regulate altitude solely through incremental pitch adjustments. When the aircraft ascends slightly due to turbulence or inertia, TECS detects an altitude deviation above the setpoint and issues a downward pitch command. Conversely, when the altitude drops below the target, an upward pitch command is issued. This cycle leads to periodic altitude oscillations of several meters around the intended flight level (see Figure 10, 20–40 seconds). Moreover, the parameter `PTCH_RATE_D_FF` controls the feedforward amplification of pitch rate, providing extra control surface force based on the target rate. However, this parameter lacks proper bounds checking, allowing an attacker to assign an excessively large value. This causes small pitch adjustments to be amplified into full deflections (either maximum or minimum). When the aircraft attempts to recover from an excessive altitude, the pitch control surface may suddenly deflect fully downward (see Figure 10, after 40 seconds), leading to a catastrophic descent and ground impact while the engine remains at high throttle.

VII. Related Work

Security concerns in RVs have garnered increasing attention from researchers in recent years. Choi et al. [34] proposed a detection framework for RV attacks based on control invariants to defend against physical attacks. However, their approach is incapable of identifying bugs originating within the control software itself. RVFuzzer [5] targets input validation bugs in RV control software by employing a control instability detector, focusing on whether configuration parameter checks are incorrectly implemented or entirely missing. Similarly, LGDFuzzer [7] leverages a learning-guided search strategy to identify range constraint violations caused by the interaction of multiple configuration parameters within RV systems. These methods concentrate on inputs related to control stability, overlooking control commands, environmental inputs, and numerous configuration parameters unrelated to control, thereby limiting their ability to detect other categories of bugs. Choi et al. [6] also introduced an approach that mutates environmental factors to validate flaws in RV security check mechanisms, but this method cannot uncover vulnerabilities present in other regions of the input space. PGFuzz [8] aims to detect policy violations stemming from design flaws or incorrect implementations; however, it heavily relies on the precision of manually specified policies and is restricted to security requirements derived from RV documentation. As a result, it fails to detect bugs caused by conflicting execution logic or undocumented implementation errors.

In addition, PatchVerif [21] is designed to evaluate RV patch code to determine whether modifications successfully resolve existing issues or introduce new ones. Its scope is limited to inputs associated with the patch code itself, with the primary goal of exposing behavioral differences. In contrast, ADGFUZZ explores input subspaces to uncover bugs triggered by complex interactions among inputs.

Traditional fuzz testing techniques [35], [36], [37], [38] for conventional software primarily focus on detecting memory overflow vulnerabilities. However, for inputs that do not cause memory overflows, there are no well-defined evaluation metrics. ADGFUZZ by analyzing the assignment relationships between variables, is able to detect security issues that are independent of memory, such as falling caused by accumulated computational inaccuracies and deviations from the planned flight route.

There are other RV-related research works, including bug forensics and localization in RVs based on flight log analysis [17], [39], [16], automated patch generation and repair [40], and sensor-targeted attacks in RV systems [41], [42], [43], [44], [45], [46], [47]. These topics fall outside the scope of our work.

VIII. Discussion and Limitation

A. Temporal Gap Between Bug Activation and Detection

A subtle yet critical challenge in RV control software is the temporal gap between the triggering of a bug and its eventual detection. Unlike traditional software, where bugs often result in immediate crashes or explicit error codes, certain bugs in RV software may only manifest after a sequence of events, often with delayed or uncertain timing. A representative example arises in waypoint navigation tasks, where the vehicle is expected to follow a predefined trajectory (e.g., Fig. 1). If the RV deviates in a direction that aligns with its intended heading, the deviation might remain undetected until it reaches the designated waypoint. In such cases, the bug is triggered well before the waypoint is reached, but detection may be

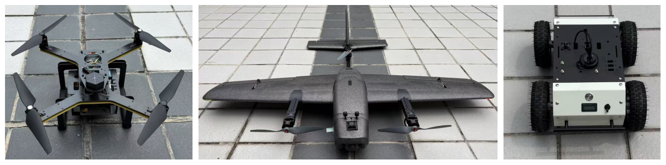


Fig. 11. Three real-world RV: (a) quadrotor (Z410), (b) fixed-wing aircraft (HeeWing-T2), and (c) unmanned ground vehicle (CQR).

significantly delayed or, in the worst-case scenario, completely missed if the testing duration expires beforehand.

To mitigate this issue, we designed navigation tasks that incorporate frequent turning maneuvers, which continually alter the RV's expected heading. This increases the chances of exposing bugs related to waypoint deviation and helps keep the temporal gap within a manageable range.

B. Limitations of SITL

SITL-based testing provides a fast, controllable, and reproducible environment for uncovering logic-level bugs in RV control software. However, SITL cannot accurately reflect hardware-specific issues such as sensor noise, actuator failures, hardware-induced latency, or electrical faults. Integrating ADGFUZZ with Hardware-in-the-Loop (HITL) testing [48], [49] is a potential solution, but such approaches require significant hardware resources and engineering overhead. Importantly, all bugs identified by ADGFUZZ are unrelated to hardware-specific issues. We validated these bugs on three different types of real-world RVs (Figure 11). Among the 87 bugs discovered, 45 can be reproduced on real-world RVs, while 42 are simulator-only, caused by SITL-specific parameters (detailed in §IX-D). For example, `SIM_PLD_ENABLE` can trigger a crash in simulation but has no effect on real RVs, as such inputs are unavailable outside SITL. Nevertheless, these bugs are still valuable, as they expose weaknesses in control logic and may indicate potential risks when similar logic is reused in flight-critical modules. Moreover, we note that ADGFUZZ cannot test assignment statements that are inactive in simulation but exercised in real-world scenarios.

C. Portability ADGFUZZ to Other Platforms

Our approach relies on extracting assignment dependencies and associating RV inputs with source code variables based on naming semantics. However, if most variable names lack semantic meaning or the source code is entirely unavailable, our approach becomes inapplicable. In such scenarios, the static analysis phase fails to construct meaningful ADGs, thereby hindering MIS extraction and entropy-aware fuzzing. We acknowledge this limitation and emphasize that ADGFUZZ is primarily designed for open-source, semantically rich flight control software such as ArduPilot [9] and PX4 [50].

For standardized and open-source RV platforms, users can adapt ADGFUZZ to other systems by following five steps: (1) Construct the ADG from the RV software's source code.

(2) Identify user-controllable RV inputs to infer MISs from the ADGs. (3) Examine the semantic mapping between ADG variables and RV inputs. Identify missing but semantically similar (variable term, input term) pairs and update the term association tables accordingly. Then regenerate MISs. (4) Modify the input injection and feedback processing modules according to differences in the communication protocol, to support fuzzing execution and oracle judgment. (5) Finally, eliminate false positives by examining whether the input's intended functionality inherently causes the observed behavior (see §IX-A). The effort depends on the similarity between the target RV software and ArduPilot. For example, PX4 shares the same naming conventions with ArduPilot, allowing us to skip Step (1). We spent approximately one hour on Steps (2), (3), and (5). During this time, we updated the term association table with two new data sets specifically for PX4. Although PX4 also uses MAVLink as its communication protocol, it differs in implementation; we spent 7 hours modifying 173 lines of code to address these differences. Bugs found in PX4 are described in §IX-E.

IX. Conclusion

We propose ADGFuzz, a novel fuzzing framework specifically designed to detect assignment statement bugs in RV control software. ADGFuzz identifies data dependencies between assignment statements and RV multi-source inputs by exploiting naming similarities. It subsequently applies entropy-aware fuzzing over the derived RV inputs to improve the efficiency of bug detection. In our evaluation, ADGFuzz uncovered 87 unique bugs across three RV types, 78 of which were previously unknown. All found bugs were responsibly disclosed to the developers, and 16 have been confirmed for fixing.

Acknowledgment

We thank the anonymous reviewers for their constructive comments and valuable suggestions, which helped improve the quality of this paper. This project is supported by National Natural Science Foundation of China (Grant No.92467201).

Ethics Considerations

We have responsibly reported all discovered bugs to the developers of the RV software and collaborated with them to facilitate timely remediation. The attacks described in the case studies were conducted strictly for research purposes and were validated only in simulation environments.

References

[1] Aerial Photography, 2025. [Online]. Available: https://www.wgpdigital.com/aerial-photography/

[2] Aerial Light Show, 2025. [Online]. Available: https://www.ehang.com/formation/

[3] Drone Formation Performance, 2025. [Online]. Available: https://en.hg-fly.com/lightshow.html

[4] The world's best military drones in 2025 and their capabilities, 2025. [Online]. Available: https://shorturl.at/3VF9a

[5] 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)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 425–442. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/kim

[6] H. Choi, S. Kate, Y. Aafer, X. Zhang, and D. Xu, "Cyber-physical inconsistency vulnerability identification for safety checks in robotic vehicles," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 263–278. [Online]. Available: https://doi.org/10.1145/3372297.3417249

[7] 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*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 462–473. [Online]. Available: https://doi.org/10.1145/3510003.3510084

[8] H. Kim, M. O. Ozmen, A. Bianchi, Z. B. Celik, and D. Xu, "PGFUZZ: policy-guided fuzzing for robotic vehicles," in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/pgfuzz-policy-guided-fuzzing-for-robotic-vehicles/

[9] ArduPilot, 2025. [Online]. Available: https://ardupilot.org/

[10] SITL Simulator (Software in the Loop), 2025. [Online]. Available: https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html

[11] ArduPilot Copter, 2025. [Online]. Available: https://ardupilot.org/copter/

[12] ArduCopter Complete Parameter List, 2025. [Online]. Available: https://ardupilot.org/copter/docs/parameters.html

[13] A. Abdallah, M. Z. Ali, J. Mišić, and V. B. Mišić, "Efficient security scheme for disaster surveillance uav communication networks," *Information*, vol. 10, no. 2, p. 43, 2019.

[14] S. Lee and K. Kim, "Grand theft drone: Reaching breaking point in drone proprietary rf link security," in *Black Hat Europe 2022*, London, UK, 2022, technical briefing.

[15] Y. Mekdad, A. Acar, A. Aris, A. El Fergougui, M. Conti, R. Lazzeretti, and S. Uluagac, "Exploring jamming and hijacking attacks for micro aerial drones," in *ICC 2024 - IEEE International Conference on Communications*, 2024, pp. 1939–1944.

[16] C. Lee, D. Kim, G. Kim, S. Lee, and T. Kim, "LTA: control-driven UAV testing and bug localization with flight record decomposition," in *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems, SenSys 2024, Hangzhou, China, November 4-7, 2024*, J. Liu, Y. Shu, J. Chen, Y. He, and R. Tan, Eds. ACM, 2024, pp. 450–463. [Online]. Available: https://doi.org/10.1145/3666025.3699350

[17] T. Kim, C. H. Kim, A. Ozen, F. Fei, Z. Tu, X. Zhang, X. Deng, D. J. Tian, and D. Xu, "From control model to program: Investigating robotic aerial vehicle accidents with MAYDAY," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 913–930. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/kim

[18] P. Moosbrugger, K. Y. Rozier, and J. Schumann, "R2u2: monitoring and diagnosis of security threats for unmanned aerial systems," *Formal Methods in System Design*, vol. 51, pp. 31–61, 2017.

[19] ArduPilot Style Guide, 2025. [Online]. Available: https://shorturl.at/BFKnX

[20] ArduPilot parameter naming guidelines, 2025. [Online]. Available: https://shorturl.at/JFhCU

[21] H. Kim, M. O. Ozmen, Z. B. Celik, A. Bianchi, and D. Xu, "PatchVerif: Discovering faulty patches in robotic vehicles," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 3011–3028. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/kim-hyungsub

[22] ArduPilot Style Guide-Commenting, 2025. [Online]. Available: https://shorturl.at/u4Lie

[23] C. E. Shannon, "A mathematical theory of communication," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 5, no. 1, p. 3–55, Jan. 2001. [Online]. Available: https://doi.org/10.1145/584091.584093

[24] ArduCopter Commands, 2025. [Online]. Available: https://shorturl.at/n7vLW

[25] ArduPilot Code, 2025. [Online]. Available: https://github.com/ArduPilot/ardupilot

[26] ArduPlane Complete Parameter List, 2025. [Online]. Available: https://ardupilot.org/plane/docs/parameters.html

[27] Rover Complete Parameter List, 2025. [Online]. Available: https://ardupilot.org/rover/docs/parameters.html

[28] PYMAVLink, 2025. [Online]. Available: https://github.com/ArduPilot/pymavlink

[29] A. A. Laghari, A. K. Jumani, R. A. Laghari, and H. Nawaz, "Unmanned aerial vehicles: A review," *Cognitive Robotics*, vol. 3, pp. 8–22, 2023.

[30] ArduPilot Plane, 2025. [Online]. Available: https://ardupilot.org/plane/

[31] ArduPilot Rover, 2025. [Online]. Available: https://ardupilot.org/rover/

[32] MAVProxy, 2025. [Online]. Available: https://ardupilot.org/mavproxy/

[33] MAVLink, 2025. [Online]. Available: https://mavlink.io/en/

[34] H. Choi, W.-C. Lee, Y. Aafer, F. Fei, Z. Tu, X. Zhang, D. Xu, and X. Deng, "Detecting attacks against robotic vehicles: A control invariant approach," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 801–816. [Online]. Available: https://doi.org/10.1145/3243734.3243752

[35] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu, "Context-sensitive and directional concurrency fuzzing for data-race detection," 01 2022.

[36] D. She, A. Shah, and S. S. Jana, "Effective seed scheduling for fuzzing with graph centrality analysis," *2022 IEEE Symposium on Security and Privacy (SP)*, pp. 2194–2211, 2022.

[37] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "GREYONE: Data flow sensitive fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2577–2594. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/gan

[38] libFuzzer – a library for coverage-guided fuzz testing, 2025. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[39] H. Choi, Z. Cheng, and X. Zhang, "Rvplayer: Robotic vehicle forensics by replay with what-if reasoning," 01 2022.

[40] H. Kim, M. O. Ozmen, Z. B. Celik, A. Bianchi, and D. Xu, "Pgpatch: Policy-guided logic bug patching for robotic vehicles," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 1826–1844.

[41] H. Kim, R. Bandyopadhyay, M. O. Ozmen, Z. B. Celik, A. Bianchi, Y. Kim, and D. Xu, "A systematic study of physical sensor attack hardness," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 2328–2347.

[42] D. Davidson, H. Wu, R. Jellinek, T. Ristenpart, and V. Singh, "Controlling uavs with sensor input spoofing attacks," in *Proceedings of the 10th USENIX Conference on Offensive Technologies*, ser. WOOT'16. USA: USENIX Association, 2016, p. 221–231.

[43] J. Jang, M. Cho, J. Kim, D. Kim, and Y. Kim, "Paralyzing drones via EMI signal injection on sensory communication channels," in *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023.

[44] J. Jeong, D. Kim, J. Jang, J. Noh, C. Song, and Y. Kim, "Un-rocking drones: Foundations of acoustic injection attacks and recovery thereof," in *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023.

[45] K. C. Zeng, S. Liu, Y. Shu, D. Wang, H. Li, Y. Dou, G. Wang, and Y. Yang, "All your GPS are belong to us: Towards stealthy manipulation of road navigation systems," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1527–1544. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/zeng

[46] H. Sathaye, M. Strohmeier, V. Lenders, and A. Ranganathan, "An experimental study of GPS spoofing and takeover attacks on UAVs," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3503–3520. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/sathaye

[47] Y. Xu, X. Han, G. Deng, J. Li, Y. Liu, and T. Zhang, "Sok: Rethinking sensor spoofing attacks against robotic vehicles from a systematic view," in *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2023, pp. 1082–1100.

[48] ArduPilot HITL Simulators, 2025. [Online]. Available: https://ardupilot.org/dev/docs/hitl-simulators.html

[49] PX4 Hardware in the Loop Simulation, 2025. [Online]. Available: https://docs.px4.io/v1.12/en/simulation/hitl.html

[50] PX4 Open Source Autopilot, 2025. [Online]. Available: https://px4.io

[51] C. Romeo, "Developers dislike security: Ten frustrations and resolutions," Presentation at RSA Conference 2021, May 2021.

[52] R. Mohanani, I. Salman, B. Turhan, P. Rodríguez, and P. Ralph, "Cognitive biases in software engineering: A systematic mapping study," *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1318–1339, 2020.

[53] jMAVSim. Simple multirotor simulator with MAVLink protocol support, 2025. [Online]. Available: https://github.com/PX4/jMAVSim

APPENDIX

A. RV-Inherent False Positives

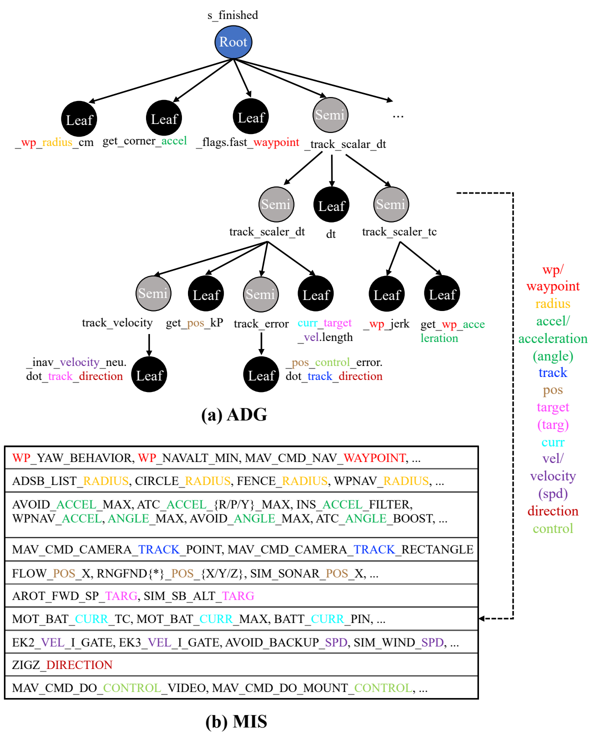
We list several inputs below that cause false positives. The configuration parameters `MOT_PWM_MAX` and `SIM_ENGINE_MUL` affect motor output; setting them too low or to zero leads to insufficient thrust or disabled throttle, causing the vehicle to descend uncontrollably. Similarly, `SIM_GPS1_ENABLE` disables GPS simulation when set to zero, triggering EKF failsafe and switching to `LAND` mode. The commands `MAV_CMD_NAV_LAND` and `MAV_CMD_COMPONENT_ARM_DISARM` also cause intentional descent or immediate motor shutdown, which may be misclassified as "crash on the ground" by the oracle. Additionally, combining `MAV_CMD_DO_CHANGE_SPEED` to reduce velocity with a high wind speed (`SIM_WIND_SPD`) can result in route deviation if the drone lacks sufficient forward motion. Excessively large values for `SIM_GPS1_POS_X`, `POS_Y`, and `POS_Z` may cause abnormal GPS jumps, leading to false "route deviation" reports. Modifying `SYSID_THISMAV` can break GCS communication, triggering a false "software crash" detection. These input combinations were manually excluded during result analysis by one of the authors.

B. An Example ADG–MIS Pair

Figure 12 shows the ADG constructed from the code snippet in Figure 2, along with the corresponding MIS derived from that ADG. For clarity, some data elements have been omitted. Variables and RV input terms with semantic associations are highlighted using the same color.

C. Exploration Behavior of Entropy-Based Fuzzing Strategy

Thorough exploration of the input space remains a fundamental challenge in fuzzing. To quantify the exploration behavior of our entropy-based strategy and assess whether it introduces biased exploration or misses corner cases, we stratified all MISs into four entropy intervals in descending order: A – (75%, 100%], B – (50%, 75%], C – (25%, 50%], and D – [0%, 25%]. From each interval, we randomly selected 100 MISs and measured their cumulative code coverage after execution. Figure 13 shows a Venn diagram of coverage overlap across the four groups. High-entropy MISs generally trigger more unique paths and achieve higher overall coverage, while low-entropy MISs still expose rare paths, retaining value for corner-case discovery. The distribution of bugs across entropy intervals further confirms this trend: across three RVs, we observed 217, 140, 121, and 45 bug-triggering cases from



WP_YAW_BEHAVIOR, WP_NAVALT_MIN, MAV_CMD_NAV_WAYPOINT, ...
ADSB_LIST_RADIUS, CIRCLE_RADIUS, FENCE_RADIUS, WPNAV_RADIUS, ...
AVOID_ACCEL_MAX, ATC_ACCEL_{R/P/Y}_MAX, INS_ACCEL_FILTER, WPNAV_ACCEL, ANGLE_MAX, AVOID_ANGLE_MAX, ATC_ANGLE_BOOST, ...
MAV_CMD_CAMERA_TRACK_POINT, MAV_CMD_CAMERA_TRACK_RECTANGLE
FLOW_POS_X, RNGFND{*}_POS_{X/Y/Z}, SIM_SONAR_POS_X, ...
AROT_FWD_SP_TARG, SIM_SB_ALT_TARG
MOT_BAT_CURR_TC, MOT_BAT_CURR_MAX, BATT_CURR_PIN, ...
EK2_VEL_I_GATE, EK3_VEL_I_GATE, AVOID_BACKUP_SPD, SIM_WIND_SPD, ...
ZIGZ_DIRECTION
MAV_CMD_DO_CONTROL_VIDEO, MAV_CMD_DO_MOUNT_CONTROL, ...

Fig. 12. An illustrative ADG–MIS pair generated from a real code snippet (from Figure 2), highlighting the semantic mapping between code variables and RV inputs.

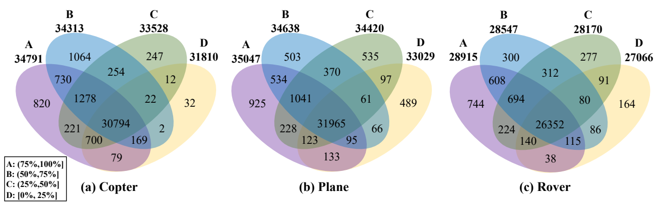


Fig. 13. Venn diagram illustrating the overlap and uniqueness of code coverage achieved by MISs from different entropy intervals.

the highest to the lowest entropy groups. Therefore, ADGFUZZ employs a hybrid strategy: prioritizing high-entropy MISs for efficiency and early bug discovery, while periodically exploring low-entropy MISs to avoid missing critical cases.

D. List of Discovered Bugs and Impact Assessment

Table IV presents a comprehensive list of all bugs discovered by ADGFUZZ in ArduPilot. For each bug, we provide: (1) the index number, (2) the affected RV type(s), (3) the input that triggers the bug, (4) a brief description of the bug, (5) the impact type reflecting its severity, (6) the reproducibility status on real RVs, and (7) whether upstream developers have considered fixing it. Among these, item (5), the type of impact, is used to characterize the nature and severity of the bug. We systematically classify impact types into three categories: security-critical, safety-relevant, and usability-related. Security-critical bugs can be triggered by external or untrusted inputs and lead to software crashes, overflows, or denial-of-service. Safety-relevant bugs are not necessarily caused by attackers; they may also result from user misconfigurations or improper usage. These bugs can cause abnormal or hazardous physical behavior during RV operation (e.g., thrust loss or crashes). Usability-related bugs mainly indicate insufficient software robustness. Although they typically do not affect the physical world and are less severe, they can reveal underlying flaws in control logic.

Among the 87 bugs identified, #1–26 are crash-to-ground, #27–41 are route deviations, and the remaining ones are software crash bugs. We classified 15 of them as security-critical, 31 as safety-relevant, and the remaining 41 as usability issues. We reproduced 45 of these bugs on real-world RVs (Figure 11), and the observed behavior was highly consistent with that in the SITL simulation environment. The bugs that could not be reproduced on physical RVs were mainly due to one of the following reasons: (1) the relevant inputs are not used by real hardware (e.g., parameters like SIM_RATE_HZ are only effective in the simulator), or (2) we were unable to recreate the specific environmental conditions required to trigger the bug (e.g., strong wind).

For the bugs that developers chose not to fix, our communication with the ArduPilot maintainers revealed two main reasons: First, repair difficulty. Developers noted that comprehensive parameter checks are impractical on resource-constrained devices due to increased code size and performance impact. Moreover, some parameters inherently lack a well-defined legal range, making it difficult to enforce a universal validation rule. For example, the minimum acceptable PWM value varies significantly across different models of Electronic Speed Controllers (ESCs): some stop at 1000, while others stop at 1150. Thus, fixed thresholds may be inappropriate or even harmful. Second, cognitive bias. Although developers acknowledged the existence of external or internal adversaries, many expressed the view that ArduPilot is merely one component of a larger system, and that security enforcement should be handled at the communication link level or by external systems. This reflects a cognitive bias in the perception of security responsibility, which downplays potential risks and shifts accountability for mitigation [51], [52]. Given these considerations, developers tend to avoid fixing such issues even when the abnormal behavior is confirmed.

E. Bugs Discovered in PX4

After applying ADGFUZZ to PX4 [50] (with jMAVSim [53] simulator), we identified 35 bugs in the quadcopter, including 5 software crashes, 26 ground crashes, and 4 route deviation bugs. Among these, 22 bugs were triggered by a single input, 10 by two inputs, 2 by three inputs, and 1 by four inputs. Unlike ArduPilot, where many bugs are associated with simulator-only parameters, the PX4 bugs are triggered by inputs that remain valid on real vehicles. Although these bugs were first observed in simulation, they stem from assignment dependencies within the same control logic executed on actual robotic vehicles, highlighting their potential real-world impact.

Table IV. Bugs Discovered by ADGFuzz in ArduPilot. We use C to denote Copter, P for Plane, and R for Rover.

Index	RV Type	Inputs	Description	Impact Type	Reproducibility	Status
1	C	`AHRS_EKF_TYPE`	Assigning special values (e.g., 0 or 11) to this parameter results in the copter crashing to the ground.	Safety	Yes	Pending
2	C	`SERVO3_FUNCTION`	Setting this parameter to special values (e.g., 141) leads to thrust loss and the copter crashes.	Safety	Yes	Pending
3	C	`MOT_BAT_CURR_MAX`, `MOT_PWM_MIN`	Setting either parameter individually does not induce issues; however, concurrently assigning both parameters to small values results in complete loss of thrust and a crash.	Safety	Yes	Yes
4	C	`FLTMODE_CH`	Assigning a special value (e.g., 8) to this parameter leads to thrust loss and a subsequent crash.	Safety	Yes	Pending
5	C	`AHRS_EKF_TYPE, AHRS_TRIM_X`	Assigning these two parameters to 2 and the minimum legal value (-0.1745), respectively, causes the copter to crash to the ground.	Safety	Yes	Pending
6	C	`SIM_GPS1_LAG_MS`	Setting to a large value leads to unstable flight attitude,leading to a crash after a short period.	Usability	No	No
7	C	`SIM_TIME_JITTER,MAV_CMD_DO_SET_MODE`	Causes the copter to crash to the ground.	Safety	No	No
8	C	`MOT_BAT_VOLT_MAX`, `MOT_PWM_MIN`	Setting either parameter independently does not cause adverse effects; however, assigning both parameters simultaneously results in thrust loss and a crash.	Safety	Yes	Yes
9	C	`FS_THR_VALUE`	Setting to the maximum legal value (1100) prevents the safety mechanism from triggering, causes the copter to crash.	Safety	Yes	Pending
10	C	`SIM_RATE_HZ`	Assigning values within a specific range (e.g., 85) causes the copter to flip unstably and fall to the ground.	Usability	No	No
11	C	`MAV_CMD_NAV_LOITER_UNLIM`	Normal use of this command unexpectedly changes the copter's flight mode, causing it to hit the ground.	Safety	Yes	Yes
12	C	Multiple `MOT_SPIN_MIN`	During waypoint navigation, setting `MOT_SPIN_MIN` to 0.6 causes the copter to climb after reaching the waypoint. Restoring it to the nominal 0.15 corrects the altitude. Setting it to 0.6 again results in a brief climb followed by a stall and crash.	Safety	Yes	Yes
13	P	`SIM_SERVO_SPEED`	Inadequate validation of parameter ranges allows a value of 100, which results in the plane crashing.	Usability	No	No
14	P	`TECS_PITCH_MIN`, `PTCH_RATE_D_FF`	When the mission waypoint altitude is set to 30 meters, assigning large values to these parameters causes the plane to first climb to approximately 50 meters, then rapidly descend and crash.	Safety	Yes	Pending
15	P	`RCMAP_PITCH`	Assigning special values (e.g., 10 or 16) causes the plane to gradually lose altitude and ultimately crash.	Safety	Yes	Pending
16	P	`AIRSPEED_MIN`	If the plane cannot reach the target speed under current parameter settings, it lowers its pitch and altitude to accelerate, which may lead to gradual descent and eventual crash.	Safety	Yes	Pending
17	P	`ARSPD_OFFSET`	No proper range checking; large values cause the plane to crash to the ground.	Safety	Yes	Pending
18	P	`TECS_SPD_OMEGA`	Large values cause linear speed reduction and eventual crash.	Safety	Yes	Pending
19	P	`SIM_SERVO_DELAY`	Small values cause the plane to flip in flight and rapidly crash.	Usability	No	No
20	P	`PTCH2SRV_RMAX_UP`	Assigning small values causes the plane to crash.	Safety	Yes	Pending
21	P	`ICE_ENABLE`	Out-of-range values incorrectly cause the plane to crash to the ground.	Safety	Yes	Pending
22	P	`TECS_TIME_CONST`	Assigning large values leads to deceleration and eventual crash.	Safety	Yes	Pending
23	P	`PTCH2SRV_RLL`	Assigning large values causes the plane to overshoot the prescribed waypoint altitude and oscillate at high altitudes; if the waypoint altitude is below 20 meters, the plane crashes.	Safety	Yes	Pending
24	P	`AIRSPEED_STALL`	Setting to large values causes the plane to crash to the ground.	Safety	Yes	Pending
25	C&P	`SIM_CAN_SRV_MSK`	Assigning special values (e.g., 10 or 30) causes the plane to lose power and crash.	Usability	No	Pending
26	C&P	`BARO_ALT_OFFSET`	Assigning large values results in the plane descending during flight and crashing.	Safety	Yes	Pending
27	C	`MOT_PWM_MIN`	When set to the maximum value, the copter rapidly ascends and deviates from waypoints. After some distance, it erroneously marks waypoints as reached, eventually recording all waypoints as completed despite not having actually reached them.	Safety	Yes	Yes
28	C	`MOT_SPIN_MIN`	Same as above.	Safety	Yes	Yes
29	C	`SIM_GPS1_LAG_MS`	Lack of proper value constraints allows large values to cause the copter to deviate from the planned flight path.	Usability	No	No
30	C	`ANGLE_MAX, SIM_WIND_SPD`	With a small `ANGLE_MAX` (e.g., 1600) and high wind speed, the copter cannot complete turns due to tilt angle limitations.	Safety	Yes	Yes
31	C	`SIM_WIND_TURB`	Values above 10 cause attitude oscillation and speed fluctuations, leading to flight path deviation.	Usability	No	No
32	C	`AHRS_EKF_TYPE, AHRS_TRIM_X`	Changing both parameters simultaneously causes the copter to deviate from its planned route and eventually crash.	Safety	Yes	Pending
33	P	`RCMAP_ROLL`	Assigning a special value (e.g., 6) to this parameter causes the plane to abort the mission and immediately enter a loitering state.	Safety	Yes	Pending
34	P	`SIM_SERVO_SPEED`	Assigning a low value to this parameter results in the plane lacking a fixed trajectory and deviating from the planned route.	Usability	No	No
35	R	`ATC_SPEED_FF`	Assigning the maximum value to this parameter causes the rover to deviate from the planned route and, after a period, incorrectly report waypoint completion even when distant from the waypoint (Similar to No.27).	Safety	Yes	Yes
36	R	`SIM_CAN_SRV_MSK`	Setting the third bit to 1 (e.g., 0100 or 1110) causes the rover to rapidly move in the direction opposite to the intended route.	Usability	No	Pending
37	R	`BATT_ARM_MAH`, `SIM_TIME_JITTER`	When both parameters are assigned large values, the rover fails to recognize arrival at the target waypoint, instead lingering nearby.	Usability	No	No
38	R	`ATC_STR_RAT_FF`	Assigning large values stalls the rover near the target, ending the navigation task.	Safety	Yes	Pending
39	R	`FRAME_CLASS`	Assigning large values stalls the rover near the target, ending the navigation task.	Safety	Yes	Pending
40	R	`FENCE_RADIUS, FENCE_ENABLE`	When these parameters are assigned after the rover has moved, it detects a fence violation but fails to trigger RTL, causing the vehicle to freeze instead of returning to the home position.	Safety	Yes	Pending
41	R	`ATC_SPEED_D_FF`	Assigning large values causes the rover to slow to zero speed near a waypoint, then quickly reverse course, repeatedly approaching and departing the waypoint in a loop.	Safety	Yes	Pending
42	C	`SIM_BATT_VOLTAGE`	Assigning large values to this parameter results in a floating-point exception and subsequent software crash.	Usability	No	No
43	C	`SIM_TIME_JITTER`, `SIM_RATE_HZ`	Setting `SIM_TIME_JITTER` to a large value with a low `SIM_RATE_HZ` causes a software crash after approximately two seconds of delay.	Usability	No	No
44	C	`MAV_CMD_NAV_LOITER_TO_ALT`	Excessive x/y/z values cause overflow and crash the flight control software.	Security	Yes	Yes
45	P	`SIM_ACC_FILE_RW`	When set to the special value 3, a delay of a few seconds results in "EOF on TCP stack", after which SITL becomes unresponsive.	Usability	No	Pending
46	R	`AHRS_EKF_TYPE`	Assigning the special value 2 triggers a floating-point exception, causing a rover software crash.	Security	Yes	Pending
47	R	`INS_LOG_BAT_LGCT`	Assigning the special value 0 triggers a floating-point exception, resulting in a rover software crash.	Security	Yes	Pending
48	R	`SIM_WIND_T_ALT`	Assigning the special value 0 triggers a floating-point exception, resulting in a rover software crash.	Usability	No	No
49	R	`SIM_MAG_ALY_HGT`	Assigning the special value 0 triggers a floating-point exception, resulting in a rover software crash.	Usability	No	No
50	R	`SIM_CAN_SRV_MSK`	Assigning special values 7 or 21 causes the rover to spin erratically, followed by a software crash due to floating-point overflow after one to two seconds.	Usability	No	No
51	C&P	`SIM_RATE_HZ`	Setting this value too low (0–22) triggers a floating-point exception and results in a software crash.	Usability	No	No
52	C&P	`GPS2_TYPE`	Assigning the special value 11 results in a segmentation fault and software crash.	Security	Yes	Pending
53	C&P	`SIM_WIND_TURB`	Assigning a large value to this parameter results in a floating-point exception and subsequent software crash.	Usability	No	No
54	C&P	`TERRAIN_SPACING`	Setting this parameter to a value less than 1 (sub-meter accuracy) leads to floating-point overflow and software crash.	Security	Yes	Yes
55-57	C&P	`SIM_IMU_POS_X/SIM_IMU_POS_Y/ SIM_IMU_POS_Z`	Assigning large values to this parameter results in a floating-point exception and subsequent software crash.	Usability	No	No
58-62	C&P&R	`INS_GYROFFS_X/INS_GYROFFS_Y/ INS_GYR2OFFS_X/INS_GYR2OFFS_Y/ INS_GYR2OFFS_Z`	Assigning large values to this parameter results in a floating-point exception and subsequent software crash.	Security	Yes	No
63	C&P&R	`SIM_DRIFT_SPEED`	Assigning large values to this parameter results in a floating-point exception and subsequent software crash.	Usability	No	No
64-69	C&P&R	`SIM_ACC1_BIAS_X/SIM_ACC1_BIAS_Y/ SIM_ACC1_BIAS_Z/SIM_ACC2_BIAS_X/ SIM_ACC2_BIAS_Y/SIM_ACC2_BIAS_Z`	Assigning large values to this parameter results in a floating-point exception and subsequent software crash.	Usability	No	No
70-75	C&P&R	`SIM_GYR1_BIAS_X/SIM_GYR1_BIAS_Y/ SIM_GYR1_BIAS_Z/SIM_GYR2_BIAS_X/ SIM_GYR2_BIAS_Y/SIM_GYR2_BIAS_Z`	Assigning large values to this parameter results in a floating-point exception and subsequent software crash.	Usability	No	No
76,77	C&P&R	`SIM_GPS1_GLTCH_X/SIM_GPS1_GLTCH_Y`	Assigning large values to this parameter results in a floating-point exception and subsequent software crash.	Usability	No	No
78,79	C&P&R	`SIM_GPS1_HZ/ SIM_GPS2_HZ`	Assigning the special value 0 triggers a floating-point exception and results in a software crash.	Usability	No	No
80	C&P&R	`SIM_PLD_ENABLE`	Assigning the special value 1 triggers a "PANIC" event, rendering the flight control software unresponsive.	Usability	No	Yes
81	C&P&R	`SIM_SONAR_SCALE`	Assigning the special value 0 triggers a floating-point exception and results in a software crash.	Usability	No	No
82	C&P&R	`SERIAL_PASS2`	Assigning special values 0 or 3 causes the software to become unresponsive and the UAV to display a "no link" message.	Security	Yes	Pending
83	C&P&R	`SIM_GYR_FILE_RW`	Assigning the special value 1 or 3 triggers a "PANIC" event, resulting in unresponsive flight control software.	Usability	No	Yes
84	C&P&R	`MAV_CMD_CAN_FORWARD`	Setting a large first value (e.g., `[498, 0, 0, 0, 0, 0, 0]`) triggers a segmentation fault and crashes the software.	Security	Yes	Yes
85	C&P&R	`MAV_CMD_DO_SET_HOME`	Supplying excessively large values in the x/y/z data fields (e.g., `[x, x, x, x, 399, 300, 699]`) causes overflow.	Security	Yes	Yes
86	C&P&R	`MAV_CMD_DO_SET_ROI_LOCATION`	Supplying excessively large values for x/y/z position data leads to overflow and subsequently causes software crashes.	Security	Yes	Yes
87	C&P&R	`MAV_CMD_EXTERNAL_POSITION_ESTIMATE`	Supplying excessively large values for x/y/z position data leads to overflow and software crashes.	Security	Yes	Yes

Appendix A
Artifact Appendix

ADGFuzz is a fuzzing framework for robotic vehicle (RV) flight control software, designed to detect three categories of bugs: software crashes, crashes to the ground, and route deviations. Our experiments were conducted on three types of RVs supported by ArduPilot: quadcopter (Copter), fixed-wing aircraft (Plane), and ground rover (Rover). This appendix provides detailed instructions on how to obtain, install, run, and verify the results of ADGFuzz.

A. Description & Requirements

1) How to access: The source code of ADGFuzz is available in a GitHub repository[2], which enables building the framework directly from source. Additionally, a pre-configured Ubuntu virtual machine (VM)[3] is provided to facilitate quick and reliable artifact evaluation. The artifact, including all necessary dependencies, datasets, and scripts, is packaged in the VM.

- **VM Credentials:**
  Username: `adgfuzz`, **Password:** `1`

2) Hardware dependencies: There are no special hardware requirements, a typical commodity computer should be sufficient to run ADGFuzz and reproduce its results.

3) Software dependencies: A Linux operating system is required. Our artifact has been tested on Ubuntu 20.04. The Python version used is 3.8.10, and all external dependencies are listed in the `requirements.txt` file.

4) Benchmarks: None.

B. Artifact Installation & Configuration

Recommended: Use the provided Ubuntu VM.

1) Import the VM, then start it and log in with the above credentials.
2) All required software and dependencies are pre-installed; you may proceed directly to the experiment workflow.

Manual installation (optional): For manual installation, please refer to the GitHub repository[2], where detailed instructions are provided.

C. Experiment Workflow

The workflow of ADGFuzz proceeds as follows:

1) **Static Analysis:** The system analyzes the RV source code to extract all assignment statements, building Assignment Dependency Graphs (ADGs). Each ADG is converted to a set of input terms, which are then mapped (via synonym and physical coupling tables) to Matched Input Subsets (MISs) accepted by the target RV control software.
2) **Input Prioritization:** The entropy of each MIS is calculated and used to guide probabilistic selection for testing.
3) **Fuzzing Phase:** For each fuzzing iteration, a fresh RV simulation instance (ArduPilot or PX4) is launched. The selected MIS is mutated to generate test cases, and three bug oracle threads monitor for runtime errors: software crash (e.g., arithmetic exception), vehicle crash (hits the ground), and route deviation (strays from planned path).
4) **Post-Processing:** All bug-triggering test cases are recorded for post-processing to identify and minimize root causes. Results are automatically deduplicated.

Quick-start ("kick-the-tires"): A minimal test case is provided for rapid validation.

Full experiment: Comprehensive testing is conducted using static analysis outputs for Copter, Plane, and Rover, with fuzzing running for up to 24 hours per configuration.

D. Major Claims

- (C1): ADGFuzz generates test cases based on the ADG constructed during the static analysis phase and fuzzes ArduPilot (A-E1).
- (C2): ADGFuzz is capable of discovering new bugs across three types of RVs (refer to Table II and Figure 5), as demonstrated by the experiments in A-E2.

E. Evaluation

1) Experiment (E1): [Quick Validation] [2 human-minutes, 5 compute-minutes]: Simplified instructions for the "Kick-the-Tires" stage.

[Preparation] Start the virtual machine, navigate to the `work/ADGFuzz` directory, and execute `source env_set.sh` to initialize the environment.

[Execution]

```
python adgfuzz.py --initfile
paths/testcopter/quicktest/ --rvtype
copter --time 300 --out_path
paths/quickresult/
```

[Results] This triggers a short fuzzing session using a minimal configuration. It repeatedly triggers a bug that causes a software crash. The resulting logs are stored in`paths/quickresult/`.

2) Experiment (E2): [Full Fuzzing Evaluation] [2 human-minutes setup, 24 compute-hours]: Using ADGFuzz to test ArduPilot's Copter module. Instructions for testing other RV types (Plane, Rover) are provided in the `README.md`[23].

[Preparation] After launching the virtual machine, navigate to the `work/ADGFuzz` directory.

[Execution] Open a terminal in the current directory and run `fuzz.sh`. The system automatically cycles through MISs and logs all detected bugs.

[Results] This script first performs static analysis (approximately one minute), then starts the fuzzing process. After some time, you can inspect the discovered bug logs under the `outfile/copter/` directory. At the end of each campaign, results are printed to the terminal and stored in log files and bug input files under `outfile/`. Log summaries report the number and type of bugs (crash, deviation, instability).

[2]https://github.com/wyunc/ADGFuzz.git
[3]https://doi.org/10.5281/zenodo.16956667