PGFUZZ: Policy-Guided Fuzzing for Robotic Vehicles

Hyungsub Kim, Muslum Ozgur Ozmen, Antonio Bianchi, Z. Berkay Celik, and Dongyan Xu
Purdue University
{kim2956, mozmen, antoniob, zcelik, dxu}@purdue.edu

***Abstract*—Robotic vehicles (RVs) are becoming essential tools of modern systems, including autonomous delivery services, public transportation, and environment monitoring. Despite their diverse deployment, safety and security issues with RVs limit their wide adoption. Most attempts to date in RV security aim to propose defenses that harden their control program against syntactic bugs, input validation bugs, and external sensor spoofing attacks. In this paper, we introduce PGFUZZ, a policy-guided fuzzing framework, which validates whether an RV adheres to identified safety and functional policies that cover user commands, configuration parameters, and physical states. PGFUZZ expresses desired policies through temporal logic formulas with time constraints as a guide to fuzz the analyzed system. Specifically, it generates fuzzing inputs that minimize a distance metric measuring "how close" the RV current state is to a policy violation. In addition, it uses static and dynamic analysis to focus the fuzzing effort only on those commands, parameters, and environmental factors that influence the "truth value" of any of the exercised policies. The combination of these two techniques allows PGFUZZ to increase the efficiency of the fuzzing process significantly. We validate PGFUZZ on three RV control programs, ArduPilot, PX4, and Paparazzi, with 56 unique policies. PGFUZZ discovered 156 previously unknown bugs, 106 of which have been acknowledged by their developers.**

I. INTRODUCTION

Robotic Vehicles (RVs) are becoming widespread both in industrial and consumer environments [7], [35], [60]. Unfortunately, RVs face diverse threats including (1) physical external attacks such as sensor spoofing attacks [61], [65], (2) software crashes due to floating-point exceptions or memory corruption issues, (3) insider attacks [4], [34], and (4) misimplementations causing safety and functional issues, which leads to undesired behaviors in the RV. Previous efforts at fuzzing have introduced techniques to address (1), (2), and (3), but (4) has not received much attention. RVs must respect safety and security policies to avoid creating physical damage to the environment in which they operate or to themselves. For instance, RVs are often equipped with a parachute. Due to safety concerns, RV's software must check preconditions to safely release the parachute (e.g., the RV must be high enough when deploying the parachute). However, the control software's careless design may allow the RV to release the parachute without checking these preconditions.

Such safety violations might lead to catastrophic consequences as reported in recent news [17], [63]. For instance, Tesla's autopilot software failed to initiate an emergency brake maneuver [63], and the Boeing-737 Max airplanes crashed because their software improperly allowed them to activate the anti-stall system [17].

Unfortunately, previous fuzzing approaches cannot discover this type of violations for the following two reasons. First, they do not consider the entire *input space* of the RV's control software, including user commands, configuration parameters, and environmental factors. Second, they only focus on finding memory corruption bugs or RV's control stability issues. Therefore, they cannot detect safety policy violations, e.g., a drone is deploying the parachute at a too-low altitude.

We develop PGFUZZ, a policy-based fuzzing framework designed to address these challenges. PGFUZZ includes three interconnected components: (1) Pre-Processing, (2) Policy-Guided Fuzzing, and (3) Bug Post-Processing.

In the Pre-Processing component, we express the correct operation of an RV through policies denoted by a metric temporal logic (MTL). Thereafter, we minimize the fuzzing space via finding inputs related to the tested policies that, when mutated, could potentially trigger policy violations. For example, given a policy in natural language stating that "the *fail-safe mode* must be triggered when the engine *temperature* is higher than 100°C", PGFUZZ expresses this policy with the MTL formula: $\square\ \{(\texttt{temperature} > \texttt{100°C}) \rightarrow (\texttt{failsafe} = \texttt{on})\}$. It then decomposes this formula into the *temperature* and the *fail-safe mode* states, and identifies fuzzing inputs such as user commands (e.g., increasing *temperature*) and configuration parameters (e.g., units of *temperature*), influencing the policy states.

Then, the Policy-Guided Fuzzing mutates inputs identified by the Pre-Processing component. It implements two kinds of distance metrics, propositional distances to guide the mutation engine, and a global distance to detect when a policy violation occurs. The distance metrics quantify how close the current system states are to a policy violation. Positive distances indicate the policy holds, whereas negative distances indicate the policy is violated. Therefore, PGFUZZ mutates inputs to minimize the global distance. After each input is sent to the control software, which runs in an RV simulator, PGFUZZ collects the system states and computes the distance metrics. The input's impact on the distance metric (whether it increases or decreases) is leveraged to decide on the next inputs. When the global distance becomes negative, a policy violation is detected. Turning to the *fail-safe mode* example, PGFUZZ mutates inputs to increase the temperature to be larger than `100°C`, and checks whether, at the same time, the *fail-safe* mode is activated.

The last component, Bug Post-Processing, minimizes the input sequence triggering the bugs by excluding inputs irrelevant to the policy violation. The minimized input sequence is then used to identify the root cause of each violated policy.

To verify the correctness and effectiveness of PGFUZZ, we

Network and Distributed Systems Security (NDSS) Symposium 2021
21-24 February 2021, San Diego, CA, USA
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24096
www.ndss-symposium.org

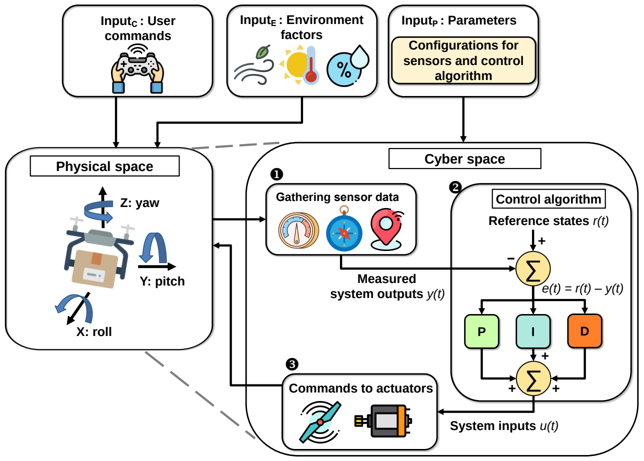


Fig. 1: Workflow of RV's control software.

used PGFuzz to fuzz ArduPilot, PX4, and Paparazzi, the three most popular flight control software packages used in many commodity RVs [10], [32], [52]. PGFuzz found 156 previously unknown bugs in 48 hours[1]. Out of the 156 bugs, the developers confirmed 106 bugs, and nine bugs have already been patched. We compared PGFuzz's results with those from previous approaches designed to find bugs in RVs. We found that 128 out of 156 found bugs can only be discovered by PGFuzz.

In summary, this paper makes the following contributions:

- **Behavior-aware Bug Oracle.** We identify policies that define RVs' safety and functional requirements and formally represent them via temporal logic formulas. PGFuzz leverages the identified policies to find bugs allowing the violation of these policies.
- **Policy-Guided Mutation Engine.** PGFuzz follows a novel fuzzing design that optimizes its bug search by (i) mutating the inputs/parameters, trying to negate the identified security policies and using, as a heuristic, dedicated distance metrics, and (ii) minimizing the fuzzing space of the inputs and parameters related to the analyzed policies.
- **Evaluation in real-world RVs.** We applied PGFuzz to the three most popular vehicle control software packages, and we discovered 156 previously unknown bugs, 106 of which have been acknowledged by developers of the affected packages.

To foster research on this topic, we make PGFuzz publicly available (https://github.com/purseclab/PGFUZZ).

II. Background

Inputs and Outputs of RVs. A vehicle leads to incorrect operation or failure when the system maintains an undesired state. For instance, a vehicle crashes to the ground when it maintains incorrect roll, pitch, and yaw angles. RVs often periodically follow three steps (See Figure 1) for their correct operation: (1) the control algorithm reads system outputs $\mathtt{y(t)}$ measured by the sensors (from ❶ to ❷), (2) the algorithm first computes errors $\mathtt{e(t)}$ based on $\mathtt{r(t)}-\mathtt{y(t)}$ where $\mathtt{r(t)}$ and $\mathtt{t}$ denote reference states and current time, and (3) a Proportional–Integral–Derivative (PID) control algorithm derives system inputs $\mathtt{u(t)}$ through $\mathtt{e(t)}$ (❸).

[1]We made responsible disclosure to the developers of the flight control software.

RVs mainly operate with three types of inputs, configuration parameters ($\mathtt{Input_P}$), user commands ($\mathtt{Input_C}$), and environment factors ($\mathtt{Input_E}$). (1) $\mathtt{Input_P}$ allows users to configure many aspects of how RVs operate. For instance, $\mathtt{K_P}$, $\mathtt{K_i}$, and $\mathtt{K_d}$ of the PID control algorithm denote tuning parameters for the proportional, integral, and derivative terms. RVs specify ranges for configuration parameters of $\mathtt{K_P}$, $\mathtt{K_i}$, and $\mathtt{K_d}$ to safely tune the PID control algorithm. (2) $\mathtt{Input_C}$ enables the users to dynamically operate RVs. The control software denies some of $\mathtt{Input_c}$ when these commands lead to an undesired system state. For example, `disarming` user command stops the vehicle's all motors, and the control software does not accept such a command while the vehicle is flying in the air. (3) $\mathtt{Input_E}$ (e.g., wind and sensor noise) also affects the system outputs $\mathtt{y(t)}$. For instance, the control software assigns a barometer sensor as a primary altitude source when GPS signals are blocked or show biased altitude values.

Fuzzing. Fuzzing is an automated testing technique that randomly or semi-randomly generates test inputs to discover bugs in programs. Existing fuzzing approaches differ in how they handle two main core aspects: input generation and bug oracle. The input generation can be completely random or guided by some heuristics. For instance, many approaches [5], [44], [48] use code coverage as a heuristic. Regarding the bug oracle, traditional fuzzing approaches use code crashes (typically caused by memory corruption) to detect inputs triggering bugs in the analyzed program. We consider these two aspects differently than in traditional, general-purpose fuzzers. Specifically, about the bug oracle, since we are dealing with RVs, we mainly aim at finding policy violations about the physical states of RVs, in addition to software crashes in the control software. We then run the control software in a simulator that is able to keep track of the physical states of the tested control software. Lastly, we define a metric measuring how "close" we are to violating one of these policies. We use this metric as a heuristic to guide our input generation.

III. Motivating Example

We provide an example of a safety issue that PGFuzz targets. ArduPilot drone control software can trigger a parachute release when it recognizes that the drone is falling to the ground with an uncontrolled attitude [10], [15]. Additionally, the user can manually trigger parachute deployment. In both cases, the ArduPilot official documentation states that the following four conditions must hold to deploy a parachute while preserving the drone safety [13]: (1) the motors must be armed, (2) the vehicle must not be in the `FLIP` or `ACRO` flight modes, (3) the barometer must show that the vehicle is not climbing, and (4) the vehicle's current altitude must be above the `CHUTE_ALT_MIN` parameter value.

Based on these requirements, we express a safety policy ($\mathtt{A.CHUTE_1}$) through metric temporal logic (MTL) (Detailed in Section V-A): $\square\{(\mathtt{Parachute}=\mathtt{on})\}\rightarrow\{(\mathtt{Armed}=\mathtt{true})\wedge(\mathtt{Mode_t}\neq\mathtt{FLIP/ACRO})\wedge(\mathtt{ALT_t}\leq\mathtt{ALT_{t-1}})\wedge(\mathtt{ALT_t}>\mathtt{CHUTE\_ALT\_MIN})\}$ where $\mathtt{t}$ and $\mathtt{ALT}$ denote time and altitude, and $\square$ is always.

Traditional fuzzing techniques targeting program crashes [5], [44], [48] clearly cannot detect such safety violations. Moreover, randomly sending commands to the ArduPilot drone simulator cannot efficiently test this policy, given the high number of commands and parameters that could be potentially mutated.

Additionally, fuzzing approaches that specifically target CPS [21], [22], [41] cannot discover this kind of safety violations for two main reasons. First, policy violations are often triggered by the composition of different types of system inputs. However, these approaches only focus on a single part of the input space, meaning they do not consider unified behavior of user commands, configuration parameters, and environmental factors. Second, their bug oracles are designed to detect specific bug types, such as deviated flight paths or instability. To detail, if a policy violation causes unexpected physical behavior, e.g., failing to trigger a GPS fail-safe mode, their bug oracles cannot detect such undesired behavior although the failing GPS fail-safe mode leads to unexpected states with potentially disastrous consequences.

To address these limitations, PGFuzz uses MTL formulas to guide both its input generation and detect safety violations. Turning to the example safety policy, PGFuzz issues system inputs that trigger a mutation of the propositional variables of the formula. At the same time, it checks whether the safety policy is violated after each input generation. By using PGFuzz, we found that ArduPilot improperly checks the first three requirements. This leads to a policy violation where the vehicle deploys the parachute when it is climbing, causing it to crash on the ground (Detailed in Section VII-C1).

Threat Model. We consider as in-scope for this paper both design flaws (from benign developers and users) and malicious intent (from adversaries) that can cause unsafe or undesired states (e.g., physical crashes) in RVs. Design flaws can happen due to poor parameter documentation, unexpected environmental conditions (e.g., sensor noise and wind), and buggy code. We assume that developers are benign; they, however, could misimplement or incorrectly design the system components. Furthermore, users can unintentionally cause safety issues via either sending commands at an inappropriate time or improperly changing configuration parameters.

While considering malicious actors, we assume that an adversary is aware of inputs causing policy violations and can trigger them with malicious intent. Particularly, an adversary can control an RV's three types of inputs. (1) An adversary can manipulate the *configuration parameters* of an RV by either overriding them before a flight or changing them after the drone takes off (similar to [41]). (2) An adversary can replay or spoof *user commands* sent to the RV by exploiting known vulnerabilities in the RV's communication protocol [43], [58]. (3) An adversary can manipulate the *environmental conditions* (or wait until suitable conditions are met) before conducting their attack (similar to [23], [41]). We detail the number of violations for each subset of these inputs in Section VII-B. For instance, we will show that an adversary is able to trigger 77% of the found policy violations by only changing the RV's configuration parameters.

The adversary's goal is to physically impact the RV's operations (e.g., causing a physical crash or disrupting the RV's camera) by *stealthily* triggering policy violations. We note that an adversary could also simply drop or disarm the vehicle by sending a malicious command (e.g., stopping actuators); however, these attacks are not stealthy. Particularly, such self-sabotaging inputs can be easily identified and prevented with run-time mission monitoring tools enforced by both the vehicle and ground control system [25], [46]. In contrast, policy violations triggered by sending an input that looks innocent are stealthier and more difficult to detect by monitoring tools. For these reasons, we do not consider these self-sabotaging attacks in-scope of this paper. In addition, physical sensor attacks (e.g., GPS and gyroscope spoofing) and malicious code injections are out of scope. The main reasons are (1) the root causes of sensor attacks arise in the hardware components (e.g., acoustic attacks against gyroscope [61]), rather than buggy code in the vehicle's control program, and (2) there exist effective techniques to detect sensor and code injection attacks [6], [28], [37]–[39]. Lastly, although PGFuzz is not designed to specifically find floating-point exceptions and other software crashes in the controller code, it reports them when triggered by the tested inputs.

IV. Approach Overview

In this section, we first present the design challenges of CPS fuzzing. We then provide an overview of PGFuzz.

A. *Design Challenges*

Traditional fuzzing techniques [5], [44], [48] including those for CPS [21], [22], [41] have two main limitations that prevent their adoption for policy-guided fuzzing in real-world systems. First, their bug oracles are not designed to detect undesired system states that do not cause a system crash, memory-access violation, or physical instability. To address this limitation, we implement a *Behavior-aware Bug Oracle*. Our bug oracle is aware of desired states of RVs via MTL formulas and detects if the formulas are violated while fuzzing the analyzed program. Second, the mutation engines of the traditional fuzzers cannot intelligently generate inputs for the RVs. This limitation is due to the large input space of the RVs, with tens of different parameters and commands, each of which can have a wide range of values. To address this limitation, we implement a *Policy-Guided Mutation Engine*. This engine is based on:

1) A mapping connecting each term of a policy with the inputs influencing the RV's states;
2) A distance metric measuring the "distance" between a vehicle's current states and policy violation.

The mutation engine uses these to guide the input mutations toward those more likely to generate a policy violation.

B. PGFuzz *Overview*

PGFuzz includes three interconnected components, (1) Pre-Processing, (2) Policy-Guided Fuzzing, and (3) Bug Post-Processing, as depicted in Figure 2.

Pre-Processing. In this step, we identify and formally represent the policies and reduce the large input space by eliminating the inputs that are not relevant to the identified policies.

Users and developers derive requirements in the targeted system by studying the RV documentation and evaluating the connections between assets and functional constraints that restrict the use or operation of assets [19], [20]. We then convert the identified requirements from natural language to policies expressed with MTL formulas (❷). PGFuzz next runs its profiling engine, which determines for each policy the limited set of inputs, $\texttt{Input}_{\texttt{min}}$, relevant to the target policy (i.e., the limited set of inputs that, when mutated, could potentially trigger policy violations). To achieve this, we first unwind the relationship between the configuration parameters ($\texttt{Input}_{\texttt{P}}$) and the RV physical states by (1) obtaining the data-flow graph of the vehicle through static analysis (if the RV's source code is available) and (2) analyzing the developer guide manuals [12], [53] (❸). However, using the static analysis and manuals makes it difficult to analyze the impacts of user commands ($\texttt{Input}_{\texttt{C}}$) and environmental factors ($\texttt{Input}_{\texttt{E}}$) on the states since (1) $\texttt{Input}_{\texttt{C}}$ and $\texttt{Input}_{\texttt{E}}$ indirectly impact many physical states through dependencies (e.g., wind affects almost all physical states) that cannot be captured in the source code via the static analysis,

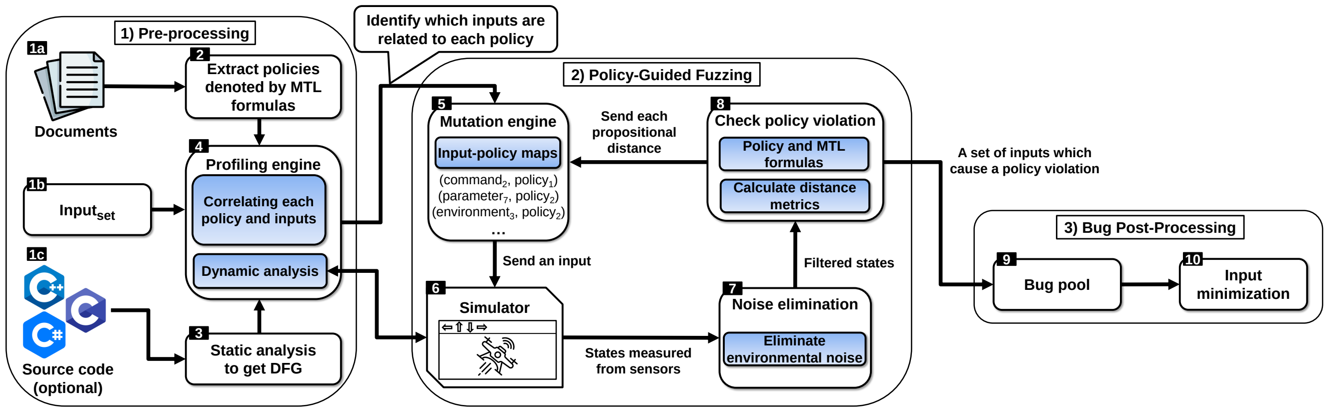


Fig. 2: Overview of PGFUZZ's workflow and architecture which consists of three components: Pre-Processing, Policy-Guided Fuzzing, and Bug Post-Processing. $\texttt{Input}_{\texttt{set}}$ includes user commands, configuration parameters, and environmental factors.

and (2) the manuals usually do not mention the impacts of $\texttt{Input}_{\texttt{C}}$ and $\texttt{Input}_{\texttt{E}}$ on the RV's states. Therefore, we conduct a dynamic analysis with an RV simulator for $\texttt{Input}_{\texttt{C}}$ and $\texttt{Input}_{\texttt{E}}$ to extract the relationships between these inputs and the RV physical states (❹). This process also enables PGFUZZ to estimate the time required for each input in the $\texttt{Input}_{\texttt{set}}$ to cause a physical effect on the RV. This information is later leveraged by PGFUZZ's mutation engine.

Policy-Guided Fuzzing. This step mutates inputs based on the computed distance metrics and uses them to fuzz the analyzed program to find policy violations. The mutation engine first selects one input among those returned by the profiling engine of the Pre-Processing component, $\texttt{Input}_{\texttt{min}}$ (❹-❺). These test inputs are sent to the simulator that runs the system and reports the physical states (e.g., sensors and actuator values) of the RV (❻-❼). During this step, we eliminate the environmental noise (e.g., vibration and wind effect) by measuring the deviation between the reference and current system states (❼). We then compute two separate "distance metrics" that formally define how close a system is to a policy violation (❽): (i) a global distance that checks whether the current system states violate a policy, and (ii) a propositional distance that intelligently mutates the inputs to lead the system closer to violations. Lastly, the mutation engine determines the particular inputs to minimize the distance. If none of the inputs decrease the distance metric, new inputs from $\texttt{Input}_{\texttt{min}}$ are selected from the input-policy maps (❺). The inputs that lead to policy violations are reported to PGFUZZ's Bug Post-Processing component (❾ ❿).

Bug Post-Processing. This step minimizes the sequence of inputs triggering the detected policy violation by excluding inputs irrelevant to the policy violation. This information is for identifying the root cause of the violated policy.

V. PGFUZZ: POLICY-GUIDED FUZZING

In this section, we detail the components of PGFUZZ, Pre-Processing (Section V-A), Policy-Guided Fuzzing (Section V-B), and Bug Post-Processing (Section V-C).

A. Pre-Processing

The Pre-Processing component aims at (1) deriving MTL formulas to express policies, and (2) building a profiling engine to narrow the fuzzing space based on MTL formulas. This allows us to obtain the minimal fuzzing space ($\texttt{Input}_{\texttt{min}}$) required for the Policy-Guided Fuzzing component (Section V-B).

1) Extracting MTL Policies: We refer to policies as the requirements that a system must satisfy for a vehicle to be considered safe. We identify the policies for RVs through requirements engineering [20] and represent the policies with formal logic that enables formal reasoning about them. The policies are expressed with Metric Temporal Logic (MTL) [1], [42]. In contrast to Linear Temporal Logic (LTL) [50] and Computation Tree Logic (CTL) [26] that enable reasoning over occurrence and event ordering, MTL extends LTL's modalities with timing constraints, which is more amenable to represent semantically rich temporal and causal relations among system states of RVs.

MTL formulas are composed of a set of atomic propositions (AP), propositional logic operators and temporal operators [42]. First, $p \in \texttt{AP}$ is a logical statement consisting of "terms". A term can be a physical state of RVs, configuration parameter, or environmental factor. Turning back to the $\texttt{A.CHUTE}_1$ policy example in Section III, ($\texttt{ALT}_{\texttt{t}} > \texttt{CHUTE\_ALT\_MIN}$) is an AP, and the $\texttt{ALT}_{\texttt{t}}$ and $\texttt{CHUTE\_ALT\_MIN}$ are terms. Second, MTL supports the propositional logic operators such as conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$). Third, the temporal operators include next ($\bigcirc_{\texttt{I}}$), always ($\Box_{\texttt{I}}$), eventually ($\Diamond_{\texttt{I}}$), and until ($\texttt{U}_{\texttt{I}}$) where $\texttt{I}$ denotes any non-empty positive interval. Formally, MTL formulas can be defined as follows: $\varphi ::= \top \mid \mathrm{p} \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \texttt{U}_{\texttt{I}} \varphi_2 \mid \bigcirc_{\texttt{I}}\varphi$, where $\mathrm{p} \in \texttt{AP}$ and $\top = \texttt{true}$.

We manually identify the policies through requirements defined in documentation and comments in the source code of popular RVs, ArduPilot, PX4, and Paparazzi. The policies are extracted in natural language and then expressed with MTL formulas. To make the policy identification process easier, PGFUZZ provides users with MTL templates to express policies as shown in Table I, similar to previous works [29], [66]. For instance, PX4's documentation that states "If time exceeds COM_POS_FS_DELAY seconds after GPS loss is detected, the GPS fail-safe must be triggered" is expressed with MTL as $\Box\{(\texttt{GPS}_{\texttt{loss}} = \texttt{on}) \rightarrow (\Diamond_{[0,\texttt{COM\_POS\_FS\_DELAY}+\texttt{k}]}\texttt{GPS}_{\texttt{fail}} = \texttt{on})\}$ (the time constraint $\texttt{k}$ is detailed in Section VII-C4).

Through this process, we identified 56 policies for our target RVs, 30 for ArduPilot, 21 for PX4, and 5 for Paparazzi (See Table XII in Appendix E). We measured the time required by a knowledgeable user to identify the policies and express them as MTL formulas. Particularly, two authors spent a total of 7.5 hours identifying ArduPilot policies, 3.5 hours for PX4, and 2.4 hours for Paparazzi. The time includes studying the target RV's official documentation/source code, writing policies in natural language,

ID	Policy Template Description	MTL Notation
T_1	$\texttt{term}_\texttt{j}$ should be true within time $\texttt{k}$ after $\texttt{term}_\texttt{i}$ is satisfied.	$\texttt{term}_\texttt{i} \rightarrow \Diamond_{[0,k]} \texttt{term}_\texttt{j}$
T_2	If $\texttt{term}_\texttt{i}$ is true, $\texttt{term}_\texttt{j}$, ... , $\texttt{term}_\texttt{n}$ are also true and $\texttt{term}_\texttt{k}$, ... , $\texttt{term}_\texttt{m}$ are false.	$\texttt{term}_\texttt{i} \rightarrow [\Box(\texttt{term}_\texttt{j} \wedge ... \wedge \texttt{term}_\texttt{n})] \wedge [\neg(\texttt{term}_\texttt{k} \wedge ... \wedge \texttt{term}_\texttt{m})]$
T_3	If $\texttt{term}_\texttt{i}$, ... , $\texttt{term}_\texttt{n}$ are true, $\texttt{term}_\texttt{j}$ is also true.	$\Box(\texttt{term}_\texttt{i} \wedge ... \wedge \texttt{term}_\texttt{n} \rightarrow \texttt{term}_\texttt{j})$

TABLE I: Policy templates that we use to express policies as MTL formulas for fuzzing.

translating them from natural language to MTL, and detecting the policy conflicts and reconciling them.

2) Profiling Engine: RVs have a large input space. For instance, ArduPilot v.4.0.3 supports 1,140 configuration parameters ($\texttt{Input}_\texttt{P}$), 58 user commands ($\texttt{Input}_\texttt{C}$), and 168 environment factors ($\texttt{Input}_\texttt{E}$). The profiling engine aims to exclude inputs unrelated to the fuzzed policies to reduce this large input space.

Figure 3 shows the six steps of the profiling engine. In the first step (❶), we map each policy into a list of terms, where each term represents a physical state of the RV, configuration parameter, or environmental factor. For example, $\texttt{A.CHUTE}_1$ policy[2] is decomposed into five terms: (1) *parachute*, *armed*, *mode*, and *altitude* are physical states of the RV and (2) `CHUTE_ALT_MIN` is a configuration parameter. We refer to this mapping as policy-term (1a).

Second, we map $\texttt{Input}_\texttt{P}$ to terms through static analysis to identify which policy terms related to $\texttt{Input}_\texttt{P}$ (❷). We refer to this mapping as parameter-term (2a). To illustrate, `ABS_PRESS` configuration parameter is an offset value for computing barometric altitude. PGFUZZ includes the `ABS_PRESS` parameter into its fuzzing input space to test $\texttt{A.CHUTE}_1$ policy because this parameter value is used to compute the *altitude* state.

Third, we derive dependencies among $\texttt{Input}_\texttt{P}$, $\texttt{Input}_\texttt{C}$, $\texttt{Input}_\texttt{E}$ via dependency analysis to infer relationships among these inputs (❸). For instance, a user desiring to deploy a parachute via `Parachute` command needs to change `CHUTE_ENABLED` configuration parameter (3a). Therefore, a dependency between `Parachute` user command and configuration parameter `CHUTE_ENABLED` is identified.

Fourth, we perform dynamic analysis with RV simulators to exclude the read-only and unsupported $\texttt{Input}_\texttt{P}$ from the constructed parameter-term map (2a). This step is crucial to reduce the parameter-term map size. We then map $\texttt{Input}_\texttt{C}$ and $\texttt{Input}_\texttt{E}$ into policy terms through dynamic analysis (❹). For example, $\texttt{A.CHUTE}_1$ policy includes *altitude* term. PGFUZZ includes the `WIND_SPEED` environmental parameter in its fuzzing space (See *Input$_E$*-term map (4b)) because it changes the *altitude* of the vehicle. Here, we exclude `ABS_PRESS` configuration parameter. Though it is related to the *altitude* term of the policy, `ABS_PRESS` is a read-only parameter (See *Input$_P$*-term map (4a)).

Fifth (❺), the profiling engine first extracts the inputs related to each policy from input-term mappings, (4a), (4b), and (4c), then, it constructs input-policy map (5a). For instance, the $\texttt{A.CHUTE}_1$ policy includes the *altitude* term. The profiling engine finds inputs related to the *altitude* including `Wind_speed` and `Parachute` in Figure 3.

Lastly, it analyzes the unknown time constraints of MTL formulas (❻). For instance, $\texttt{A.BRAKE}_1$ policy[3] is represented as $\Box\{(\texttt{Mode}_\texttt{t} = \texttt{BRAKE}) \rightarrow (\Diamond_{[0,k]}\texttt{Pos}_\texttt{t} = \texttt{Pos}_{\texttt{t}-1})\}$. To detect true positive policy violations, we obtain unknown time constraints $\texttt{k}$ by conducting dynamic analysis with the input-policy map (5a).

Mapping Each Policy onto Terms (❶). A policy is composed of the RV's physical states, configuration parameters, and environmental factors. In this step, we decompose each policy into terms, where each term is further analyzed to find the related inputs to be fuzzed (detailed below). First, we manually construct a list of physical states of the studied RVs (e.g., altitude, roll angle) through their manuals (the complete list of states is presented in Table XI Appendix A). If a policy includes one of those states, it is marked as a physical state and added to the policy-term map. Turning to $\texttt{A.CHUTE}_1$ policy, *parachute*, *armed*, *mode*, and *altitude* are all physical states and added to the policy-term map (1a). Second, a policy may contain configuration parameters ($\texttt{Input}_\texttt{P}$) and environmental factors ($\texttt{Input}_\texttt{E}$) because a vehicle's operation depends on their values. We search each term that includes $\texttt{Input}_\texttt{P}$ and $\texttt{Input}_\texttt{E}$ terms to find out whether a policy includes them. If there is a match, we similarly add these terms to the policy-term map. For instance, $\texttt{A.CHUTE}_1$ policy includes `CHUTE_ALT_MIN` configuration parameter, and no environmental factor as a term; thus, the `CHUTE_ALT_MIN` is added to the policy-term map.

Static Analysis for Narrowing Fuzzing Space (❷). The static analysis is used for identifying the terms related to each configuration parameter ($\texttt{Input}_\texttt{P}$, ❷ and (2a) in Figure 3). We use two complementary approaches to identify the related terms: (1) conducting static analysis at the LLVM intermediate representation (IR) level, and (2) parsing vehicle manuals.

First, we map each configuration parameter on the vehicle manuals to a variable in the source code. This allows us to know how the control program imports the parameters to the source code. For instance, our target control programs (i.e., ArduPilot, PX4, and Paparazzi) parses XML files containing a list of parameter names and valid ranges, then convert them to variables in the source code. Figure 4 shows how each control program accesses the imported configuration parameters. ArduPilot and PX4 store the parameters as data members of classes and access the parameters via a function call or directly access the data member. Paparazzi loads the parameters' values to the data section of memory via global variables.

Second, we build def-use chains of the identified parameter variables to map each parameter to related terms in the MTL formulas (❸ in Figure 2). We use LLVM to obtain the def-use chains defining these terms in the code. The code to load the imported parameters, which we previously identified, serves as the starting point to build these def-use chains. For scalars, we follow load and store operations recursively. For pointers, to identify data flow via pointer reference/dereference operators, we perform an inter-procedural, path-insensitive, and flow-sensitive points-to analysis [62]. More precisely, the profiling engine operates in three steps: (1) performs Andersen's pointer analysis [8] to identify aliases of the parameter variables, (2) transforms the code to its single static assignment form [59] and builds the data-flow graph (DFG), and (3) collects the def-use chain of the identified parameter variable from the built DFG.

[2] $\Box\{(\texttt{Parachute} = \texttt{on})\} \rightarrow \{(\texttt{Armed} = \texttt{true}) \wedge (\texttt{Mode}_\texttt{t} \neq \texttt{FLIP/ACRO}) \wedge (\texttt{ALT}_\texttt{t} \leq \texttt{ALT}_{\texttt{t}-1}) \wedge (\texttt{ALT}_\texttt{t} > \texttt{CHUTE\_ALT\_MIN})\}$

[3] When the vehicle is in `BRAKE` mode, it must stop within `k` seconds.

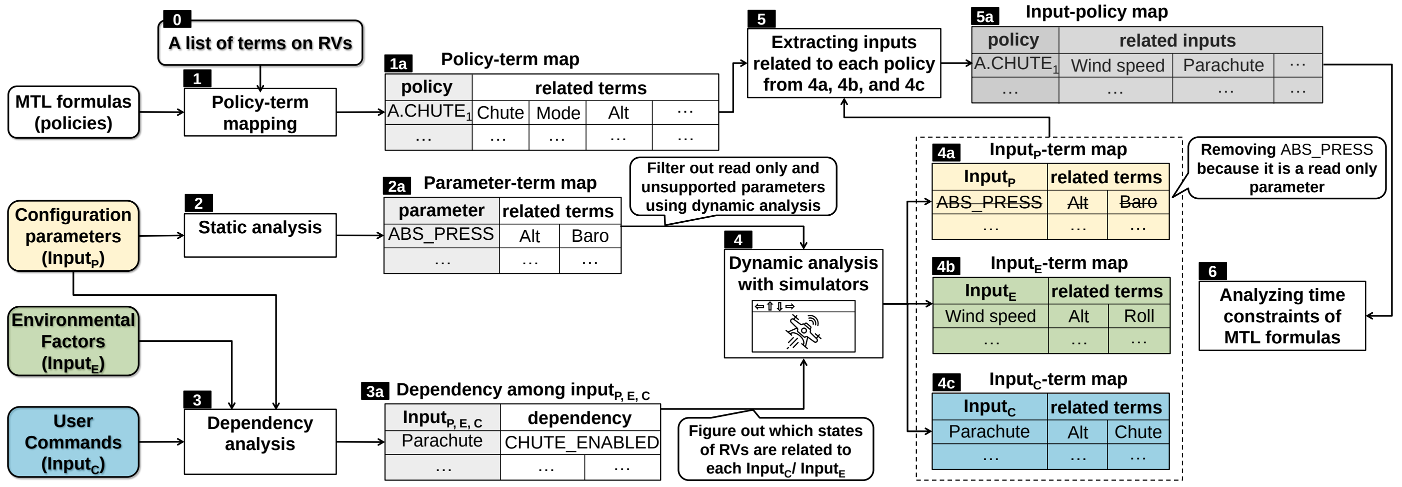
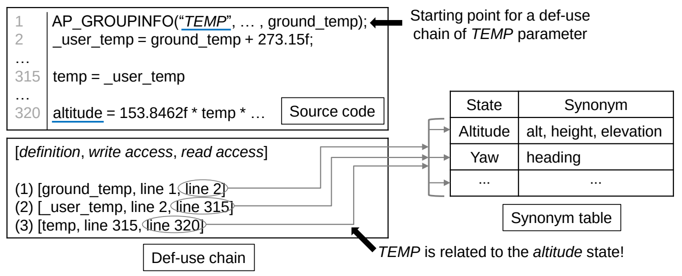

0
A list of terms on RVs
MTL formulas (policies)
1
Policy-term mapping
1a Policy-term map
policy | related terms
A.CHUTE$_1$ | Chute | Mode | Alt | …
5
Extracting inputs related to each policy from 4a, 4b, and 4c
5a Input-policy map
policy | related inputs
A.CHUTE$_1$ | Wind speed | Parachute | …
Configuration parameters (Input$_P$)
2
Static analysis
2a Parameter-term map
parameter | related terms
ABS_PRESS | Alt | Baro
Filter out read only and unsupported parameters using dynamic analysis
4
Dynamic analysis with simulators
4a Input$_P$-term map
Input$_P$ | related terms
~~ABS_PRESS~~ | ~~Alt~~ | ~~Baro~~
Removing ABS_PRESS because it is a read only parameter
4b Input$_E$-term map
Input$_E$ | related terms
Wind speed | Alt | Roll
6
Analyzing time constraints of MTL formulas
Environmental Factors (Input$_E$)
User Commands (Input$_C$)
3
Dependency analysis
3a Dependency among input$_{P, E, C}$
Input$_{P, E, C}$ | dependency
Parachute | CHUTE_ENABLED
Figure out which states of RVs are related to each Input$_C$/ Input$_E$
4c Input$_C$-term map
Input$_C$ | related terms
Parachute | Alt | Chute

Fig. 3: Profiling engine (❹ in Figure 2) steps to reduce the large input space of RVs. It outputs an input set related to each policy (the input-policy map 5a) using input-term maps (4a, 4b, and 4c) via static and dynamic analysis.

```
// ArduPilot
AP_GROUPINFO("TEMP", 3, AP_Baro, ground_temp);
// PX4
(ParamInt<px4::params::NAV_DLL_ACT>) _param_nav_dll_act;
// Paparazzi
static float phi_pgain[] = STABILIZATION_ATTITUDE_PHI_PGAIN;
static float psi_pgain[] = STABILIZATION_ATTITUDE_PSI_PGAIN;
```

Fig. 4: A code block that illustrates how the control programs access parameters after they import the parameters to their source code. The blue-colored variables represent the parameters.



```
1   AP_GROUPINFO("TEMP", … , ground_temp);
2   _user_temp = ground_temp + 273.15f;
…
315 temp = _user_temp
…
320 altitude = 153.8462f * temp * …
```

[definition, write access, read access]

(1) [ground_temp, line 1, line 2]
(2) [_user_temp, line 2, line 315]
(3) [temp, line 315, line 320]

State	Synonym
Altitude	alt, height, elevation
Yaw	heading
…	…

Starting point for a def-use chain of TEMP parameter
Source code
Def-use chain
Synonym table
TEMP is related to the altitude state!

Fig. 5: An example code block and def-use chain for illustrating the profiling engine's static analysis logic, which recursively searches the read access of the def-use chain and compares the variable name in the read access with synonyms.

Third, we manually construct a synonym table (as shown in Figure 5). This table maps source code variable names to the names used as terms by the MTL formulas. Using this table and the previously generated def-use chains, the profiling step can determine which source code variable corresponds to which term in the considered MTL formulas and, in turn, which inputs influence which internal variable (Figure 5). In this way, PGFuzz knows which inputs should be mutated to affect specific terms in the identified MTL formulas. For example, this step finds that `TEMP` configuration parameter is used for the *altitude* state because *altitude* at line 320 reads *temp*, which comes from the `TEMP` parameter value. By using this knowledge, PGFuzz mutates the `TEMP` parameter when, for instance, it tests the `A.CHUTE`$_1$ policy, since this policy includes the *altitude* state (1a in Figure 3). We manually build a synonym table for ArduPilot, PX4, and Paparazzi.

Lastly, the profiling engine parses official documentation, provided by RVs' control software developers in XML file format. This documentation describes each configuration parameter's role. It has been designed to help developers and/or users. The profiling engine first extracts all words from each parameter's description in this documentation, and then it matches the extracted words with the synonym table. For instance, ArduPilot's manual states that "it is an angle limit (to maintain altitude) time constant" to explain the `ATC_ANG_LIM_TC` parameter's role. Our profiling step matches the `ATC_ANG_LIM_TC` parameter with the *altitude* term.

Dependency among Inputs (❸). Some inputs need to be preceded by other inputs to be executed. For instance, the `Parachute` command can only be triggered if the `CHUTE_ENABLED` parameter is *true*. In this step, we first find such inputs, which cannot be effective unless another input precedes. Then, we identify which input should be executed first to execute the target input. Further, we narrow down the fuzzing inputs by eliminating user commands ($\texttt{Input}_\texttt{C}$) and configuration parameters ($\texttt{Input}_\texttt{P}$) that RV simulators do not support (❹ in Figure 3).

First, we find those $\texttt{Input}_\texttt{C}$ and $\texttt{Input}_\texttt{P}$ which cannot be effective unless another input precedes. To this end, we conduct the following steps. (1) We log all state values (e.g., altitude and roll angle) for one minute per each operation mode (e.g., `FLIP` flight mode) without any input. Then, we calculate a standard deviation of each vehicle state ($\texttt{SD}\{\texttt{State}_{(\texttt{i})}\}$). (2) We assign a random value ($\texttt{rand}_\texttt{j}$) to $\texttt{input}_\texttt{j}$ where $\texttt{input}_\texttt{j} \in \texttt{Input}_\texttt{C} \cup \texttt{Input}_\texttt{P}$, and execute it in the simulator. Specifically, we randomly assign true or false when an input requires a Boolean value, and a value within a valid range specified in the vehicle documentation when the input takes a continuous value. If the documentation does not explicitly mention the valid range, we assign a random number within $-2^{32} - 2^{32}$. (3) We then log all state values for one minute per each operation mode. We repeat these three steps 10 times and compute a standard deviation per each vehicle state ($\texttt{SD}\{\texttt{State}_{(\texttt{i,j})}\}$). After these steps are completed, we obtain two types of states: states without any input and states with the $\texttt{input}_\texttt{j}$. If the $\texttt{input}_\texttt{j}$ does not affect any state values ($|\texttt{SD}\{\texttt{State}_{(\texttt{i})}\} - \texttt{SD}\{\texttt{State}_{(\texttt{i,j})}\}| < \texttt{SD}\{\texttt{State}_{(\texttt{i})}\}$), we conclude that the $\texttt{input}_\texttt{j}$ requires another input to impact the state values.

Second, to find another input that enables $\texttt{input}_\texttt{j}$ to be activated, we conduct the following steps: (1) We select another input $\texttt{input}_\texttt{k}$ where $\texttt{input}_\texttt{k} \in \texttt{Input}_\texttt{C} \cup \texttt{Input}_\texttt{P}$. (2) We assign a random value to the $\texttt{input}_\texttt{k}$ and execute it in the simulator. (3) We make the vehicle stay in a stable state (i.e., same position and attitude). (4) We assign the previously used $\texttt{rand}_\texttt{j}$ to the $\texttt{input}_\texttt{j}$ and execute it in the simulator, while logging all state values for one minute per each operation mode. (5) We check if the $\texttt{input}_\texttt{j}$ still does not affect any state value (i.e., $|\texttt{SD}\{\texttt{State}_{(\texttt{i})}\} - \texttt{SD}\{\texttt{State}_{(\texttt{i,kj})}\}| < \texttt{SD}\{\texttt{State}_{(\texttt{i})}\}$). If that is the case, we repeat step (2) through (5) up to 10 times. If the $\texttt{input}_\texttt{k}$ enables the $\texttt{input}_\texttt{j}$ to change RV's state values (i.e., $|\texttt{SD}\{\texttt{State}_{(\texttt{i})}\} - \texttt{SD}\{\texttt{State}_{(\texttt{i,kj})}\}| > \texttt{SD}\{\texttt{State}_{(\texttt{i})}\}$), we conclude that the $\texttt{input}_\texttt{j}$ requires the $\texttt{input}_\texttt{k}$ to be executed. However, if none of the $\texttt{input}_\texttt{k}$ can enable $\texttt{input}_\texttt{j}$, we conclude that the simulators do not support that $\texttt{input}_\texttt{j}$.

For example, when $\texttt{input}_\texttt{j}$ is the `Parachute` command and $\texttt{input}_\texttt{k}$ is the `CAM_TRIGG_TYPE` parameter, which defines how to trigger a camera to take a picture, the *altitude* state values remain unchanged. This is because the `CAM_TRIGG_TYPE` parameter cannot trigger the execution of the `Parachute` command. On the other hand, if $\texttt{input}_\texttt{j}$ is the `Parachute` command and $\texttt{input}_\texttt{k}$ is the `CHUTE_ENABLED` parameter, altitude values change significantly $|\texttt{SD}\{\texttt{State}_{(\texttt{i})}\} - \texttt{SD}\{\texttt{State}_{(\texttt{i,kj})}\}| > \texttt{SD}\{\texttt{State}_{(\texttt{i})}\}$. Particularly, the `CHUTE_ENABLED` parameter enables deploying the parachute with the `Parachute` command, which impacts the `altitude`. Hence, the profiling engine identifies that the `Parachute` command is dependent on the `CHUTE_ENABLED` parameter. Therefore, for PGFUZZ to deploy the parachute, first the `CHUTE_ENABLED` parameter must be enabled, and then the `Parachute` can be sent.

Dynamic Analysis for Narrowing Fuzzing Space (❹). In this step, we analyze which states of the vehicle change according to the executed input. We first collect all state values while executing $\texttt{input}_\texttt{j}$ as described in the previous step (dependency among inputs (❸ in Figure 3)). If $|\texttt{SD}\{\texttt{State}_{(\texttt{i})}\} - \texttt{SD}\{\texttt{State}_{(\texttt{i,j})}\}| > \texttt{SD}\{\texttt{State}_{(\texttt{i})}\}$, we conclude that the $\texttt{input}_\texttt{j}$ changes the $\texttt{State}_\texttt{i}$. To illustrate, Figure 6 depicts the results of the dynamic analysis for the `throttle` user command in ArduPilot. The `throttle` user command impacts four vehicle states: *heading*, *throttle*, *altitude*, and *climb*. Figure 6 also shows that the `throttle` command affects the vehicle differently depending on the flight mode. This is because the vehicle interprets (or ignores) the `throttle` command differently based on the flight mode. For instance, to test the $\texttt{A.CHUTE}_1$ policy, PGFUZZ mutates the `throttle` command since it affects the *altitude* of the vehicle.

Extracting Inputs Related to Each Policy (❺). In this step, we first extract the inputs related to each policy from input-term mappings, 4a, 4b, and 4c. Then, we construct the input-policy map (5a). It represents a set of inputs per each policy, in which PGFUZZ will mutate those inputs to test each policy.

Analyzing Unknown Time Constraints of MTL formulas (❻). In this step, we determine the unknown time limit k in MTL formulas (e.g., when the vehicle is in `BRAKE` mode, it must stop within k seconds: $\Box\{(\texttt{Mode}_\texttt{t} = \texttt{BRAKE}) \rightarrow (\Diamond_{[0,k]}\texttt{Pos}_\texttt{t} = \texttt{Pos}_{\texttt{t}-1})\}$). This ensures the detected policy violations are true positives. To this end, we conduct the following steps: (1) We decompose the policy to terms, e.g., the $\texttt{A.BRAKE}_1$ policy (defined above) consists of two terms: *mode* and *position*. This procedure is explained in detail in ❶ - Mapping Each Policy onto Terms. (2) We randomly select one of the inputs related to the policy from the input-policy map 5a in Figure 3, assign a random value to the input, and execute it in a simulator. (3) We make the vehicle satisfy the precondition (e.g., $\texttt{Mode}_\texttt{t} = \texttt{BRAKE}$) and measure the time required (k) to satisfy the desired states (e.g., $\texttt{Pos}_\texttt{t} = \texttt{Pos}_{\texttt{t}-1}$) in the simulator. (4) We negate the precondition to test again with another random input. We repeat the steps (2) to (4) 100 times, and we define k as the maximum required time. For instance, the profiling engine notices that the `BRAKE` mode requires a maximum of 12.7 seconds to stay in the same position. Therefore, PGFUZZ starts checking the policy violation after this time limit.

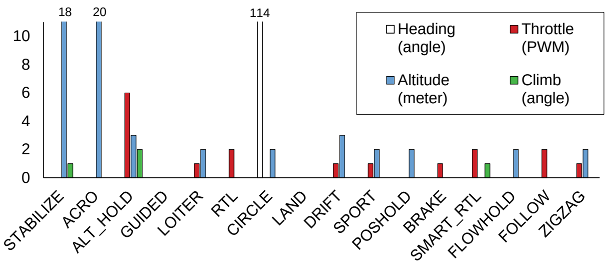


Fig. 6: Results of profiling `throttle` user command in ArduPilot. The x-axis represents each flight mode, and y-axis the $|\texttt{SD}\{\texttt{State}_{(\texttt{i,no-input})}\} - \texttt{SD}\{\texttt{State}_{(\texttt{i,throttle})}\}|$ where i denotes i-th state. The `throttle` user command changes four states: *heading*, *throttle*, *altitude*, and *climb*.

Algorithm 1 Policy-Guided Fuzzing

Input: A simulator SIM, minimized fuzzing space $\texttt{Input}_\texttt{min}$, input-policy maps MAP, an MTL formula ϕ, a fuzzing time limit τ
Output: A policy violation V and an input sequence causing the violation $\texttt{V}_\texttt{set}$

```
 1: function FUZZING(SIM, MAP, $Input_{min}$, $\phi$, $\tau$)            ▷ Main
 2:     $input_{seq} = \emptyset$                        ▷ Initialize the input sequence
 3:     while $V = \emptyset$ or $total\_time < \tau$ do
 4:         $input \leftarrow$ MUTATE(MAP, $Input_{min}$, $\phi$, DIS)   ▷ Get a mutated input
 5:         $S \leftarrow$ SIM.execute(input)       ▷ Collect RV's states (S) from SIM
 6:         $S \leftarrow$ NOISE.elimination(S)        ▷ Eliminate environmental noise
 7:         $DIS \leftarrow$ UPDATE_DISTANCE($\phi$, S)           ▷ Calculate distances
 8:         $V \leftarrow$ POLICY_CHECK($\phi$, DIS)          ▷ Check policy violations
 9:         $input_{seq} = input_{seq} \cup input$
10:     end while
11:     $V_{set} \leftarrow$ POST_BUG($input_{seq}$, V)      ▷ Conduct Bug Post-Processing
12:     return $\langle V, V_{set} \rangle$
13: end function
14: function MUTATE(MAP, $Input_{min}$, $\phi$, DIS)         ▷ Mutating inputs via MTL
15:     $input \leftarrow$ RANDOM($Input_{min}$, MAP)            ▷ Randomly pick an input
16:     $input \leftarrow$ GUIDANCE(input, $\phi$, DIS)       ▷ Pick values based on DIS
17:     return input
18: end function
```

B. Policy-Guided Fuzzing

The Policy-Guided Fuzzing component discovers policy violations given the minimized input space $\texttt{Input}_\texttt{min}$ derived at the Pre-Processing component.

1) Overview of Policy-Guided Fuzzing: Algorithm 1 details the Policy-Guided Fuzzing component's steps. The algorithm repeatedly conducts the following: (1) randomly picks an input from $\texttt{Input}_\texttt{min}$ and the input-policy maps at line 15, (2) assigns a value to the selected input based on the propositional distances at line 16. For example, when PGFUZZ increases a value of the input and it causes an increase in a propositional distance, PGFUZZ keeps assigning the same value to the input when PGFUZZ selects the input again (Section V-B4), (3) executes the mutated input on the simulator at line 5, (4) eliminates environmental noise of the physical states at line 6 (Section V-B2), (5) calculates distances according to the changed states at line 7 (Section V-B3), and (6) if PGFuzz detects a policy violation at line 8, it sends all mutated inputs ($\texttt{input}_{\texttt{seq}}$) to the Post Bug-Processing component at line 11 (Section V-C). PGFuzz repeats these steps until it detects a policy violation, or the fuzzing time is reached to a user-defined upper limit (τ).

2) Noise Elimination: In this step, we eliminate the environmental noise such as sensor noise and wind effect. Without noise elimination, PGFuzz may incorrectly guide the mutation engine (as we illustrate in Figure 7 in our evaluation). For instance, let us assume our mutation engine's goal is to minimize the altitude (to trigger a policy violation) and at time $\texttt{T}=1$, $\texttt{Alt}=15$. Then, PGFuzz executes an input to decelerate the motors' speed. This input decreases the altitude under normal conditions. However, if strong wind and sensor noise occur together with the input, the altitude increases by 2 meters, $\texttt{T}=2$, $\texttt{Alt}=17$. Then, PGFuzz wrongly determines that executing the input to decelerate the motors' speed increases the altitude. Hence, at $\texttt{T}=3$ PGFuzz increases the motors' speed, which further increases the altitude. To address this, we use reference state values in the control algorithms of flight control software. For example, ArduPilot's control algorithm keeps calculating reference altitude $\texttt{ALT}_{\texttt{ref}}$, that is the expected altitude value without noise. The reference altitude is used to compute an altitude error as $\texttt{ALT}_{\texttt{err}} = \texttt{ALT}_{\texttt{ref}} - \texttt{ALT}_{\texttt{act}}$ where $\texttt{ALT}_{\texttt{act}}$ denotes the altitude measured by sensors. Hence, $\texttt{ALT}_{\texttt{err}}$ represents the difference between the measured altitude and the expected altitude. Control programs calculate the reference states from a filtering algorithm such as the Kalman filter. However, the reference states still include noise in harsh environmental conditions (e.g., 10 m/s wind speed). To handle this problem, PGFuzz also leverages moving average to eliminate the environmental noise: $\texttt{Moving\_average}(\texttt{State\_i}_{\texttt{act}} + \texttt{State\_i}_{\texttt{err}})$ where $\texttt{State\_i}$ denotes each state (Detailed in Section VII-A2).

3) Policy Checker: The policy checker evaluates an MTL formula on the RV's states in a simulation. Given the MTL formula and the RV's physical states after executing a fuzzing input, the policy checker outputs (*i*) propositional distances to guide the mutation engine and (*ii*) a global distance to inform the bug oracle on a violated policy. The distances quantify how close a proposition and an MTL formula is to the policy violation, where negative distances indicate that a proposition or a policy is unsatisfied and positive distances indicate it is satisfied. The propositional distances are unified to derive the global distance, which indicates whether the MTL policy holds or not. The policy checker first converts the MTL formula in the *always* form into an MTL formula in *not eventually* form, and then, it arithmetically calculates the distances.

First, the policy checker converts the "imply" operator ($\rightarrow$) to the "and" operator ($\wedge$) and negates the propositions that are at the right-hand side of the "imply" operator (Detailed in Appendix B). For instance, the $\texttt{A.CHUTE}_1$ policy's MTL formula[4] is converted to $\neg\lozenge[\{(\texttt{Parachute}=\texttt{on})\}\wedge\{(\texttt{Armed}\neq\texttt{true})\vee(\texttt{Mode}_\texttt{t}=\texttt{FLIP/ACRO})\vee(\texttt{ALT}_\texttt{t}>\texttt{ALT}_{\texttt{t}-1})\vee(\texttt{ALT}_\texttt{t}<\texttt{CHUTE\_ALT\_MIN})\}]$.

Second, the policy checker derives the propositional distances as a normalized difference and uses them to compute the global distance [29], [64]. We define the propositional distances as a positive value if the proposition is true, and a negative value if it is false. Particularly, if the terms in a proposition are binary (e.g., $\texttt{Parachute}=\texttt{on}$), the distance is set to 1 when the proposition is satisfied and -1 when it is not. In contrast, a numerical distance is computed as a normalized difference (e.g., $(\texttt{CHUTE\_ALT\_MIN} - \texttt{ALT}_\texttt{t})/\texttt{CHUTE\_ALT\_MIN}$) when the terms in a proposition are numeric (e.g., $(\texttt{ALT}_\texttt{t} > \texttt{CHUTE\_ALT\_MIN})$). The propositional distances ($\texttt{P}_1 - \texttt{P}_5$) of the parachute policy are:

$$(1)\ \texttt{P}_1 = \begin{cases} 1 & \text{if } \texttt{Chute}_\texttt{t} = \texttt{on} \\ -1 & \text{if } \texttt{Chute}_\texttt{t} \neq \texttt{on} \end{cases}$$

$$(2)\ \texttt{P}_2 = \begin{cases} 1 & \text{if } \texttt{Armed}_\texttt{t} \neq \texttt{true} \\ -1 & \text{if } \texttt{Armed}_\texttt{t} = \texttt{true} \end{cases}$$

$$(3)\ \texttt{P}_3 = \begin{cases} 1 & \text{if } \texttt{Mode}_\texttt{t} = \texttt{FLIP/ACRO} \\ -1 & \text{if } \texttt{Mode}_\texttt{t} \neq \texttt{FLIP/ACRO} \end{cases}$$

$$(4)\ \texttt{P}_4 = \frac{\texttt{ALT\_t} - \texttt{ALT}_{\texttt{t}-1}}{\texttt{ALT\_t}}$$

$$(5)\ \texttt{P}_5 = \frac{\texttt{CHUTE\_ALT\_MIN} - \texttt{ALT}_\texttt{t}}{\texttt{CHUTE\_ALT\_MIN}}$$

We compute the global distance based on the propositional distances. Particularly, the arithmetic global distance is derived by converting "not", "and", "or" to -1, $\texttt{min}$, and $\texttt{max}$ [29], [64]. The $\texttt{A.CHUTE}_1$'s global distance is $-1\times[\texttt{min}\{\texttt{P}_1,\texttt{max}(\texttt{P}_2,\texttt{P}_3,\texttt{P}_4,\texttt{P}_5)\}]$. To automatically generate code snippets to compute the propositional and global distances, the policy checker first creates a binary expression tree based on the converted MTL formula in *not eventually* form, then, it traverses the nodes of the tree (a detailed example is given in Appendix B). Lastly, the policy checker flags a policy violation when the global distance becomes negative.

4) Mutation Engine: The mutation engine feeds inputs to the simulator to minimize the global distance, where the negative values of the global distance indicate a policy violation. We notice that maximizing the propositional distances (and making them positive) results in minimizing the global distance (and making it negative) since the propositional distances are negated while computing the global distance. Turning back to the $\texttt{A.CHUTE}_1$ policy, the global distance $-1\times[\texttt{min}\{\texttt{P}_1,\texttt{max}(\texttt{P}_2,\texttt{P}_3,\texttt{P}_4,\texttt{P}_5)\}]$ is negative when the propositional distances are positive. Hence, the mutation engine conducts the following steps to maximize the propositional distances. (1) It first randomly selects an input from the $\texttt{Input}_{\texttt{min}}$ of the target policy, which is stored in the input-policy map. Then, it randomly selects a value and assigns it to the input. (2) It executes the selected input on the simulator, computes the propositional and global distances, and flags a policy violation if the global distance becomes negative. (3) If the executed input increases the propositional distance, the mutation engine stores the input with the assigned value. For instance, if the input is increasing the altitude, it contributes to increasing the propositional distance $\texttt{P}_4$. The mutation engine stores this input-value pair ($\texttt{altitude,increase}$). (4) When the mutation engine randomly selects the stored input again (e.g., changing altitude), it applies the stored value to the input instead of randomly assigning a new value to the input. For instance, it executes ($\texttt{altitude,increase}$) to keep increasing the propositional distance. The mutation engine repeats the step (1)-(4) until it finds a policy violation. If PGFuzz cannot find a policy violation, it stops fuzzing when the input sequence's size is more than 1,000 or fuzzing time is more than 2 hours. Then, PGFuzz changes the target policy to another one.

5) Working Example: Table II shows an example of how our Policy-Guided Fuzzing works. The example focuses on the $\texttt{A.CHUTE}_1$ policy. When $\texttt{T}=1$, the mutation engine (❺ in Figure 2)

[4] $\square\{(\texttt{Parachute}=\texttt{on})\} \rightarrow \{(\texttt{Armed} = \texttt{true}) \wedge (\texttt{Mode}_\texttt{t} \neq \texttt{FLIP/ACRO})\wedge(\texttt{ALT}_\texttt{t}\leq\texttt{ALT}_{\texttt{t}-1})\wedge(\texttt{ALT}_\texttt{t}>\texttt{CHUTE\_ALT\_MIN})\}$

Time (T)	Parachute (on/off)	Armed (on/off)	FLIP/ACRO mode (true/false)	Measured altitude (m)	Filtered altitude (m)	p1	p2	p3	p4	p5	Global distance $(-1 \times [\mathtt{min}\{P_1,\mathtt{max}(P_2,P_3,P_4,P_5)\}])$	Fuzzed input for next time T
1	off	on	false	92	94	-1	-1	-1	0	0.06	1	WIND_SPEED = 5
2	off	on	false	93	95	-1	-1	-1	0.01	0.05	1	Parachute = on
3	off	on	false	95	95	-1	-1	-1	0	0.05	1	Increase throttle
4	off	on	false	97	99	-1	-1	-1	0.04	0.01	1	WIND_SPEED = 5
5	off	on	false	102	104	-1	-1	-1	0.05	-0.04	1	Parachute = on
6	on	on	false	106	106	1	-1	-1	0.02	-0.06	-0.02	-

TABLE II: Propositional and global distances guided with $\mathtt{Input_C}$, $\mathtt{Input_P}$, and $\mathtt{Input_E}$ (CHUTE_ALT_MIN is set to 100 meters).

randomly selects an input from the identified $\mathtt{Input_{min}}$ (❹). It chooses WIND_SPEED parameter and assigns a random value (i.e., 5) to the parameter. When $\mathtt{T}=2$, PGFUZZ calculates actual altitude based on the deviation between reference and current altitudes from ❻. PGFUZZ predicts the actual altitude is 95 meters instead of 93 meters (❼), eliminating the noise. The mutation engine first notices that the changed wind speed increases $\mathtt{P_4}$ but it decreases $\mathtt{P_5}$. Then, it stores the input-value pair, i.e., (WIND_SPEED,5) which will be used if the mutation engine randomly selects the WIND_SPEED parameter again. The mutation engine randomly chooses a user command (i.e., releasing a parachute command). However, ArduPilot does not deploy the parachute at $\mathtt{T}=3$ because the current altitude (i.e., 95 meters) is less than CHUTE_ALT_MIN which is 100 meters as default. PGFUZZ increases the throttle value, increasing altitude as the next input at $\mathtt{T}=3$. At $\mathtt{T}=4$, the global distance is the same as the previous one. However, it increases the $\mathtt{P_4}$ propositional distance due to the increased altitude. Therefore, the mutation engine also stores the pair of input and value, i.e., (throttle,increase). It first randomly selects the WIND_SPEED parameter again as the next input, then, it assigns the stored value 5 to the parameter. At $\mathtt{T}=5$, PGFUZZ randomly selects releasing a parachute command as the next input. At $\mathtt{T}=6$, the policy violation checker (❽) detects a policy violation because the parachute is deployed while the vehicle is climbing, which violates $\mathtt{A.CHUTE_1}$.

C. Bug Post-Processing

PGFUZZ conducts Bug Post-Processing to find the minimized sequence of inputs that causes a policy violation. The minimized sequence can be later used to analyze the violation's root cause. The Bug Post-Processing consists of the bug pool (❾ in Figure 2) and input minimization steps (❿ in Figure 2).

The bug pool first stores the violated policy ($\mathtt{policy_V}$) with an input sequence that causes the policy violation. The input sequence consists of each pair of input and mutated value $(\mathtt{input_i},\mathtt{value_i})$ where $\mathtt{input_i} \in \mathtt{Input_{min}}$. The input sequence includes all inputs and values from the start of fuzzing a policy until finding a policy violation. Therefore, it might contain some inputs which do not contribute to the policy violation i.e., the same policy violation can be triggered without executing some of the inputs. For example, the input sequence $\{(\mathtt{mode}=\mathtt{ACRO}),(\mathtt{wind}=5),(\mathtt{parachute}=\mathtt{on})\}$ violates $\mathtt{A.CHUTE_1}$ policy[5]. However, $(\mathtt{wind}=5)$ does not contribute to the policy violation.

Second, to find the inputs that contribute to the policy violation, the input minimization step operates as follows: (1) It creates a new process to execute a separate simulator. (2) It creates a new input sequence by excluding an input $(\mathtt{input_i})$ from the original input sequence that caused the violation $(\mathtt{input_{(1,\ldots,n)}})$. For instance, it excludes $(\mathtt{parachute}=\mathtt{on})$ from the input sequence $\{(\mathtt{mode}=\mathtt{ACRO}),(\mathtt{wind}=5),(\mathtt{parachute}=\mathtt{on})\}$. (3) It executes $(\mathtt{input_{(1,\ldots,i-1,i+1,\ldots,n)}},\mathtt{value_{(1,\ldots,i-1,i+1,\ldots,n)}})$ on the simulator (e.g., $\{(\mathtt{mode}=\mathtt{ACRO}),(\mathtt{wind}=5)\}$). (4) If the new input sequence does not lead to the same policy violation, PGFUZZ notices that the pair of $\mathtt{input_i}$ and $\mathtt{value_i}$ is mandatory to violate the policy. For example, the $\mathtt{A.CHUTE_1}$ policy cannot be violated without $(\mathtt{parachute}=\mathtt{on})$. We repeat from step (2) to (4) until the input minimization step finds a minimized input sequence which still causes the same policy violation. Turning back to the $\mathtt{A.CHUTE_1}$ policy example, the minimized input sequence is $\{(\mathtt{mode}=\mathtt{ACRO}),(\mathtt{parachute}=\mathtt{on})\}$.

[5] $\square\{(\mathtt{Parachute}=\mathtt{on})\} \rightarrow \{(\mathtt{Armed} = \mathtt{true}) \wedge (\mathtt{Mode_t} \neq \mathtt{FLIP/ACRO}) \wedge (\mathtt{ALT_t} \leq \mathtt{ALT_{t-1}}) \wedge (\mathtt{ALT_t} > \mathtt{CHUTE\_ALT\_MIN})\}$

Users can easily perform a root cause analysis based on the minimized input sequence with a violated policy. For instance, they can identify a missing flight mode check from $\{(\mathtt{mode}=\mathtt{ACRO}),(\mathtt{parachute}=\mathtt{on})\}$. We provide such examples when we introduce the case studies in Section VII-C1.

VI. IMPLEMENTATION

We evaluate PGFUZZ on the three most popular flight control software, ArduPilot, PX4, and Paparazzi as target RV controllers.

Simulator Configuration. All of the three flight control software use MAVLink [47] as their communication protocol between the flight control software and Ground Control Stations (GCSs). However, each flight control software implements the MAVLink protocol differently. To deploy PGFUZZ on ArduPilot and PX4, we choose Pymavlink v2.4.9 library [56] and PPRZLINK v2.0 library [51] for Paparazzi. Their libraries allow PGFUZZ to control vehicles through MAVLink v2.0.

Static Analysis. We choose the Low Level Virtual Machine (LLVM) 9.0.0 [45] to convert source code of the three flight control software to *bitcode*, the intermediate representation (IR) of LLVM. To obtain data flow graphs (DFG), we use a Static Value-Flow Analysis tool [62]. We wrote 386 lines of code (LoC) in C to collect all def-use chains of $\mathtt{Input_P}$ and correlate each state and $\mathtt{Input_P}$. To map names of variables on source code to names of states on policies, we manually construct a variable name mapping table (i.e., synonym table Figure 5) for each flight control software.

Dynamic Analysis. We write 586 LoC in Python using Pymavlink APIs for ArduPilot. We modify 54 LoC to integrate it into PX4. We also write 741 LoC in Python for PPRZLINK as Paparazzi uses a different library than others.

Mutation Engine. We write a total 1,379 LoC in Python for mutation engine, noise elimination, and policy checking. We modify 94 LoC of 1,379 LoC for PX4 as they differently implement MAVLink. We write 1,830 LoC for Paparazzi.

Bug Post-Processing. We use Pymavlink and PPRZLINK APIs to implement the Bug Post-Processing component. We write 626 LoC in Python for ArduPilot and PX4, and 794 LoC for Paparazzi.

VII. EVALUATION

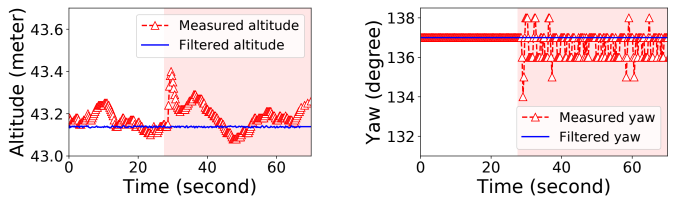
We first evaluate how each component of PGFUZZ contributes to the overall fuzzing effectiveness (Section VII-A). We then

RV system	# of $\mathtt{Input_P}$	After static analysis filtering	After dynamic analysis filtering	% of reduced target input space
ArduPilot	1,140	268	209	18.3 %
PX4	579	333	176	30.3 %
Paparazzi	82	57	51	62.2 %

TABLE III: Reduced fuzzing space for configuration parameters ($\mathtt{Input_P}$).

RV system	# of $\mathtt{Input_C}$	# $\mathtt{Input_E}$	After dynamic analysis filtering	% of reduced target input space
ArduPilot	58	168	150	66.4 %
PX4	66	30	43	44.8 %
Paparazzi	116	8	46	37.1 %

TABLE IV: Reduced fuzzing space for user commands ($\mathtt{Input_C}$) and environmental factors ($\mathtt{Input_E}$).



(a) Changed altitude values. (b) Changed yaw angles.

Fig. 7: The changed sensor values under `LOITER` flight mode with environment factors.

evaluate PGFuzz's effectiveness in finding bugs in real flight control software, ArduPilot, PX4, and Paparazzi (Section VII-B).

A. Component Evaluation

1) Profiling Engine Evaluation: Table III shows the fuzzing space reduction generated by the Pre-Processing step, relative to configuration parameters ($\mathtt{Input_P}$). The decreased fuzzing space on ArduPilot, PX4, and Paparazzi is 18.3%, 30.3%, and 62.2%, respectively. PGFuzz achieves the highest reduction rate in ArduPilot because ArduPilot includes 504 hardware configuration parameters which are irrelevant for our analysis and 21 read only parameters. On the other hand, it shows the lowest reduction rate on Paparazzi because most of its 82 configuration parameters related to attitude and altitude control algorithm (i.e., $\mathtt{K_P}$,$\mathtt{K_i}$, and $\mathtt{K_d}$ of the PID control algorithm Section II) have a direct effect on the drone's behavior. Table IV shows reduced fuzzing space for user commands ($\mathtt{Input_C}$) and environmental factors ($\mathtt{Input_E}$). The decreased fuzzing space on ArduPilot, PX4, and Paparazzi is 66.4%, 44.8%, and 37.1%, respectively.

2) Noise Elimination: Figure 7 shows the results of the noise elimination component (❼ in Figure 2) on ArduPilot. We record the sensor values every 10 ms and use 10 m/s wind speed, the wind direction of 60 degrees Z-axis, 3 m/s^2 acceleration noise, and `LOITER` flight mode[6]. We configure the width of the moving average window as 4 in the noise elimination component. As shown in Figure 7, it filters out the changes in attitude due to the noise and wind. The areas with a red background in the figure indicate when the wind is enabled.

B. Framework Evaluation

To evaluate the effectiveness of PGFuzz, we integrate it into ArduPilot, PX4, and Paparazzi, to find safety and security policy violations. Table V presents the software version and subject

[6]In `LOITER` mode, the flight control software automatically maintains the current location, heading (i.e., yaw), and altitude.

RV system	Version	Subject Vehicle	Simulator
ArduPilot	4.0.3	Quadrotor	APM SITL [9] Gazebo [33]
PX4	1.9		JSBSim [18] Gazebo
Paparazzi	5.16		NPS [49] Gazebo

TABLE V: Fuzzing target RVs.

ID	Description
A.ALT_HOLD$_2$ PP.HOVER$_Z$	If the throttle stick is in the middle (i.e., 1,500) the vehicle must maintain the current altitude.
A.FLIP$_1$	If and only if roll is less than 45 degree, throttle is greater or equal to 1,500, altitude is more than 10 meters, and the current flight mode is one of `ACRO` and `ALT_HOLD`, then the flight mode can be changed to `FLIP`.
A.GPS.FS$_1$	When the number of detected GPS satellites is less than four, the vehicle must trigger the GPS fail-safe mode.
A.LOITER$_1$ PX.HOLD$_1$ PP.HOVER$_C$	The vehicle must maintain a constant location, heading, and altitude
A.CHUTE$_1$	Parachute can be deployed only when the following conditions are satisfied: (1) the motors must be armed, (2) the vehicle must not be in the `FLIP` or `ACRO` flight modes, (3) the barometer must show that the vehicle is not climbing, and (4) the vehicle's current altitude must be above the `CHUTE_ALT_MIN` parameter value.
A.RC.FS$_1$	If and only if the vehicle is armed in `ACRO` mode and the throttle input is less than the minimum (`FS_THR_VALUE` parameter), the vehicle must immediately disarm.
A.RC.FS$_2$	If the throttle input is less than `FS_THR_VALUE` parameter, it must change the current mode to the RC fail-safe mode.
ALIVE	The vehicle must keep sending heartbeat messages to ground control systems every `k` seconds (this policy applies to `A`/`PX`/`PP`).
PX.GPS.FS$_1$	If time exceeds `COM_POS_FS_DELAY` seconds after GPS loss is detected, the GPS fail-safe must be triggered.
PX.GPS.FS$_2$	If the GPS fail-safe is triggered and a remote controller is available, the flight mode must be changed to `ALTITUDE` mode.
PX.TAKEOFF$_1$	When the vehicle conducts a taking off command, the target altitude must be the `MIS_TAKEOFF_ALT` parameter value.

TABLE VI: Example policies violated by ArduPilot (`A`), PX4 (`PX`) and Paparazzi (`PP`) (See Appendix E for complete list of policies).

vehicles used in our evaluation. We run PGFuzz for 48 hours using Ubuntu 18.04 64-bit running on an Intel Core i7-7700 CPU @ 3.6 GHz with 32 GB of RAM.

We identify policies based on their documents and represent them in MTL formulas. Table VI presents example identified policies. However, we found that each flight control software has a different level of detail in its documentation. For instance, ArduPilot provides detailed documentation that explains its correct operations, including its intended behavior in each flight mode and fail-safe logic. In contrast, Paparazzi does not provide appropriate documentation that explains its correct operation. To extract policies from Paparazzi, we used developer comments in the source code, which details the correct behavior of vehicles in different flight modes, and we converted them to MTL formulas.

Our evaluation results are shown in Table VII. PGFuzz found a total of 156 bugs. Some policy violations are caused by multiple bugs. For instance, A.ALT_HOLD$_2$ was violated by either a broad valid range of parameters or GPS failure (Detailed in Section VII-C2). Moreover, some bugs cause multiple violations, e.g., repeatedly activating the `FLIP` mode makes the vehicle crash on the ground. This violates both A.FLIP$_1$ and A.ALT_HOLD$_2$ policies. We group found bugs into four categories. (1) "Broad valid range" bugs mean that valid ranges of configuration parameters are set incorrectly. For example, `ATC_RATE_R_MAX` has a valid range from 0 to 1080. However, when users assign less than 100, the vehicle leads to unstable attitude control and crashes on the ground. (2) "Misimplementation" bugs happen when a feature does not

Policy	# of bugs	Root cause				Physical effect			
		Broad valid range	Misimplementation	Unimplemented	No checking valid range	Crash on the ground	Software crash	Unstable attitude	Unexpected behavior
$\texttt{A.ALT\_HOLD}_2$	7	2	5			4			3
$\texttt{A.FLIP}_1$	1		1						1
$\texttt{A.FLIP}_1$ $\texttt{A.ALT\_HOLD}_2$	1		1			1			
$\texttt{A.GPS.FS}_2$	1		1						1
$\texttt{A.LOITER}_1$	8	2	1		5	1		4	3
$\texttt{A.CHUTE}_1$	1		1						1
$\texttt{A.RC.FS}_1$	1		1						1
$\texttt{A.RC.FS}_2$	1		1						1
A.ALIVE	82		5		77		82		
Total (ArduPilot)	**103**	4	17	0	82	6	82	4	11
PX.ALIVE	8			8			8		
$\texttt{PX.GPS.FS}_1$	2		2						2
$\texttt{PX.GPS.FS}_2$	2		2						2
$\texttt{PX.HOLD}_1$	23	20	1		2	9		13	1
$\texttt{PX.TAKEOFF}_1$	1		1						1
Total (PX4)	**36**	20	6	8	2	9	8	13	6
$\texttt{PP.HOVER}_C$	10	10				4		6	
$\texttt{PP.HOVER}_Z$	7	7				1		2	4
Total (Paparazzi)	**17**	17	0	0	0	5	0	8	4
Total (all)	**156**	**41**	**23**	**8**	**84**	**20**	**90**	**25**	**21**

TABLE VII: Summary of found 156 previously unknown bugs in the three popular flight control software. PGFuzz found 103 previously unknown bugs in ArduPilot, 36 in PX4, and 17 in Paparazzi. (The policy descriptions are given in Appendix E.)

work properly either under normal or in a particular situation after developers implement the feature. For instance, PX4 fails to trigger a GPS fail-safe mode in specific flight modes. (3) "Unimplemented" bugs refer to unimplemented sensor failure handling conditions though these are mentioned in their documents. We found such bugs, particularly in PX4. (4) "No checking valid range" bugs mean that valid ranges of configuration parameters are not checked. For instance, a vehicle yields a floating-point exception when the `ATC_RATE_R_MAX` parameter is assigned to a value out of its predefined range. The identified policy violations cause different undesired behaviors in the vehicles, as shown in the Table VII "Physical effect" column. We divide the physical effects of the bugs into four categories. (1) "Crash on the ground" represents the vehicle that loses its attitude control and then sends a free fall warning message to the GCS. (2) "Software crash" happens when the flight control software crashes due to a floating-point exception. (3) "Unstable attitude" represents a vehicle having a fluctuating attitude. (4) "Unexpected behavior" represents all the other issues, including non-checking preconditions to change vehicle states (Section VII-C1), failing to stay at the same altitude (Section VII-C2), wrongly calculated altitude after acrobatic flying (Section VII-C3), and failing to trigger a fail-safe mode (Section VII-C4).

Analysis of Bugs. We refer to the "misimplemented" and "unimplemented" categories in Table VII as logic bugs. Out of 156 bugs, PGFuzz detected 31 (19.9%) logic bugs. The "broad valid range" and "no checking valid range" bugs involve input validation and memory safety bugs. We consider an input validation bug as a memory safety bug when it causes memory corruption. PGFuzz detected 90 (57.7%) memory safety bugs and 35 (22.4%) input validation bugs. Lastly, we refer to the identified bugs as harmless when they do not cause a crash, unstable attitude, and incorrect altitude. For instance, assigning wrongly converted angles to a camera gimbal does not lead to any operational effect on the vehicle. PGFuzz detected 11 (7.05%) harmless bugs out of 156 bugs.

False Positives. We found a set of input combinations cause false positives in the violated policies. For instance, when PGFuzz assigns zero to `SIM_ENGINE_MUL` parameter in ArduPilot, the simulator turns off the vehicle's engine, which leads to a policy

Input types causing bugs	# of bugs
$\texttt{Input}_P$	120
$\texttt{Input}_C$	10
$\texttt{Input}_E$	2
$\texttt{Input}_P$ and $\texttt{Input}_C$	20
$\texttt{Input}_P$ and $\texttt{Input}_E$	3
$\texttt{Input}_P$, $\texttt{Input}_C$, and $\texttt{Input}_E$	1
Total	**156**

TABLE VIII: Required input types to trigger bugs. $\texttt{Input}_P$, $\texttt{Input}_C$, and $\texttt{Input}_E$ represent configuration parameters, user commands, and environmental factors.

violation. We exclude such inputs ($\texttt{Input}_C$, $\texttt{Input}_P$, and $\texttt{Input}_E$) from our analysis. Further, we found that a vehicle might crash on the ground and violate policies while PGFuzz operates the vehicle too acrobatically. Thus, we also limit inputs leading the acrobatic operations (Detailed in Appendix C).

Analysis of Input Types. Each policy violation is triggered by different types of inputs. Table VIII presents the number of bugs caused by different input types. This analysis details the capabilities that an adversary requires to trigger a bug. More specifically, an adversary who can change one of the configuration parameters, user commands, or environmental factors is able to trigger 132 out of 156 bugs. Additionally, 23 bugs require changing two types of inputs, and only one bug requires control over three types of inputs.

Comparison of PGFuzz with no Policy-guided Mutation. We compare the results of PGFuzz with fuzzing without using policy guided mutation with (a) excluded input space from the profiling engine, i.e., inputs unrelated to policies, (b) full input space, and (c) reduced input space. In all cases, inputs are randomly sampled from the input space. Figure 8 compares the results of PGFuzz with (a), (b), and (c), the time vs. the cumulative number of found bugs overtime on the three systems. PGFuzz found the 156 bugs[7] in about 48 hours. The fuzzing with no policy guided mutation and the excluded input space, i.e., in the case of (a), we found only 21

[7]We note that the total number of bugs 156 does not include the 21 bugs found by the excluded input space.

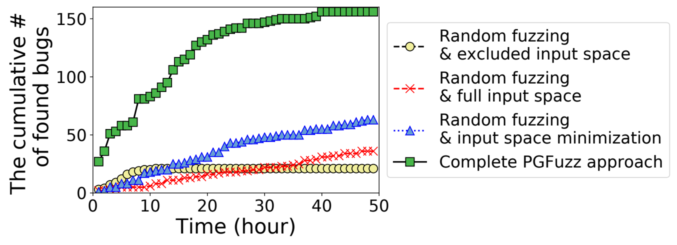


Fig. 8: Results of fuzzing based on our guidance and minimized input space in ArduPilot, PX4, and Paparazzi.

bugs, in the case of (b), we found 36 bugs, and in the case of (c), we found 63 bugs. All identified bugs in the case (a), (b), and (c) are floating-point exceptions.

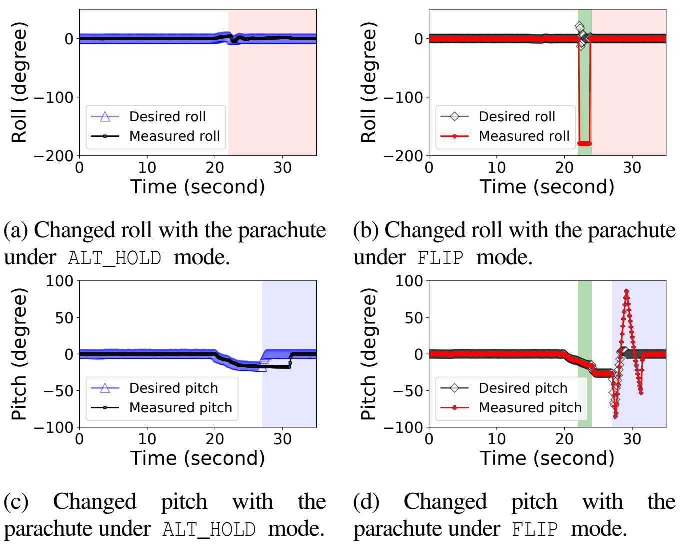
Accuracy of the Minimized Input Space. PGFUZZ reduces original input space to a minimized space related to policies (See Table III and Table IV). If many bugs are triggered by inputs excluded by the profiling engine (❹ in Figure 2), PGFUZZ may miss bugs because it does not mutate inputs from the excluded input space. However, the excluded input space leads to a total of 21 floating-point exception bugs, as shown in Figure 8. Therefore, the reduced input space is still effective at finding the bugs due to mainly two reasons. First, the excluded inputs do not affect physical states related to the policies. For instance, 132 configuration parameters in ArduPilot are for on-screen displays on a GCS; thus, they do not affect the vehicle's physical states. Second, most of the excluded inputs are self-sabotaging. For instance, GPIO pin configurations and commands for turning off actuators. While these inputs lead to abnormal behaviors, we do not consider these cases as policy violations/bugs, as discussed in our threat model (Section III).

Comparison of PGFUZZ with RVFuzzer. A recent testing system, RVFuzzer [41], discovers input validation bugs in RV control programs. Specifically, RVFuzzer only fuzzes configuration parameters and a single environmental factor wind to find control instability bugs, e.g., unstable attitude or deviation from a flight path. We contacted RVFuzzer's authors to determine how many of the bugs reported by PGFUZZ can be discovered by RVFuzzer. RVFuzzer could find 28 out of 156 bugs. Three reasons prevent RVFuzzer from detecting the 128 bugs that PGFUZZ reported. First, if a policy violation does not affect the vehicle's attitude and flight path, RVFuzzer cannot detect the violation because RVFuzzer only uses one policy that defines the stable attitude and a correct flight path. For example, the aforementioned `parachute` requires the flight control software to check some preconditions to deploy the parachute. However, these conditions, which leads to unsafe states, are not checked by RVFuzzer. Second, some bugs are only disclosed with user commands, environmental factors, and configuration parameters (Section VII-C). However, RVFuzzer only mutates inputs for the configuration parameters. Lastly, RVFuzzer cannot discover a set of bugs due to its limited binary search-based algorithm [40]. For instance, `PSC_ACC_XY_FILT` has 2.0 as default value. The vehicle does not show any unsafe state when it has 0 and 2.0. In this case, RVFuzzer concludes that [0,2] is a safe valid range. However, the vehicle leads to an unstable attitude and even crashes on the ground when PGFUZZ assigns 0.0001 to the parameter.

Responsible Disclosure. We reported the identified bugs to the vendors (development teams of RV software). 106 bugs out of the total 156 bugs have been acknowledged by developers. Table IX details the bugs for each flight control software whether they are patched/will be patched. We categorize the 106 confirmed bugs into

	# of bugs	**# of acknowledged bugs**	**# of bugs will be patched**	**# of patched bugs**
ArduPilot	103	79	5	3
PX4	36	27	21	6
Paparazzi	17	0	0	0
Total	**156**	**106**	**26**	**9**

TABLE IX: Results of responsible disclosure.



(a) Changed roll with the parachute under `ALT_HOLD` mode.

(b) Changed roll with the parachute under `FLIP` mode.

(c) Changed pitch with the parachute under `ALT_HOLD` mode.

(d) Changed pitch with the parachute under `FLIP` mode.

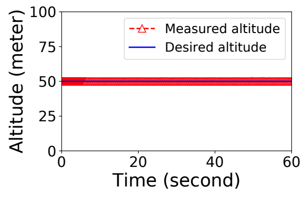
Fig. 9: The changed attitudes with the parachute. The red, green, and blue areas on the figures denote the released parachute, `FLIP` mode, and landed on the ground, respectively.

four categories based on their physical impact on the vehicle, as detailed in Table VII. (1) 6 bugs cause crashes on the ground, (2) 74 bug causes software crashes, (3) 9 bug causes unstable attitudes (i.e., unstable roll, pitch, and yaw), and (4) 17 bugs cause unexpected behaviors (i.e., deviating from assigned missions). At the time of writing, 9 software crash bugs have been patched, and 26 of the bugs are confirmed and will be patched. The remaining 71 bugs are software crash issues in ArduPilot. The root cause of the 71 bugs is that ArduPilot does not check whether some parameters are within their valid ranges (although these ranges are stated in the documentation). These missing checks lead to floating-point exceptions when they are assigned to too large, too small, or zero values. Based on the feedback we got from the developers, they stated that users are responsible for assigning values to these parameters. For now, they do not consider code updates to prevent users from assigning unsafe values to the reported parameters. They stated that if they insert every parameter check code snippet, micro-controllers' limited memory space may affect the vehicle's proper operation.

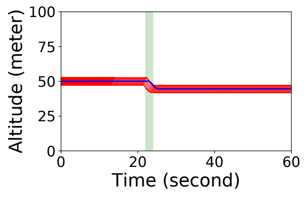
C. Case Studies

We detail four policy violations identified by PGFUZZ. We first describe the underlying reasons causing each violation and then show how attackers can exploit them to force undesired vehicle states. We note that existing RV fuzzing works cannot discover these bugs because they do not fuzz all input types and do not implement a proper bug oracle to detect them.

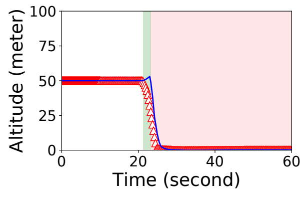
1) Case Study 1 - Unexpected Behaviors due to Misimplementation: RVs must check a set of preconditions to safely enter a new state. However, PGFUZZ discovered that the flight control software does not check the preconditions or incorrectly verify these preconditions.



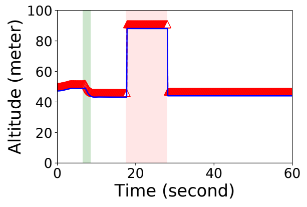
(a) Altitude during `ALT_HOLD` mode under normal conditions.



(b) Altitude during `ALT_HOLD` mode, when a flip maneuver is performed.



(c) Altitude during `ALT_HOLD` mode, when a flip maneuver is performed, and `ATC_RAT_-RLL_FF` and `ATC_RATE_R_MAX` parameters are set incorrectly.



(d) Altitude during `ALT_HOLD` mode, when the parameter `GPS_POS1_Z` is set incorrectly, and a flip maneuver is performed.

Fig. 10: Altitude values in different scenarios. The white color background shows when the `ALT_HOLD` mode is enabled. The green color shows the time the `FLIP` mode is enabled. The red color shows the time $\texttt{A.ALT\_HOLD}_2$ policy is violated.

Policy. ArduPilot documentation explicitly states the conditions to deploy a parachute: (1) the motors must be armed, (2) the vehicle must not be in the `FLIP` or `ACRO` flight modes, (3) the barometer must show that the vehicle is not climbing, and (4) the vehicle's current altitude must be above the `CHUTE_ALT_MIN` parameter value. PGFUZZ detected policy violations while checking the $\texttt{A.CHUTE}_1$ policy (See Table XII for its MTL) that defines these conditions.

Root Cause. PGFUZZ discovered that ArduPilot only checks the last condition among the four preconditions when the parachute is manually released. To illustrate, Figure 9a and Figure 9c depict the drone's attitude changes with the released parachute in the `ALT_HOLD` mode, showing that the drone performs a stable landing. However, when PGFUZZ triggers the `FLIP` mode and deploys the parachute at the same time (at 22 seconds), it loses pitch controls and then crashes on the ground at 28 seconds (See Figure 9b and Figure 9d). The flight control software sends a crash warning message to GCSs when it detects landing on the ground with an unstable attitude. We identify the drone's crash via the MAVLink message.

Attack. An attacker capable of spoofing/replaying user commands to trigger the `FLIP` mode and deploy the parachute simultaneously is able to cause a crash. We note that though the attacker triggers the parachute, this action still looks like an innocent command because the flight control software can automatically deploy the parachute without the manual command when it determines that the drone is losing attitude control. To prevent such unsafe state transitions, ArduPilot requires to check the four conditions before deploying the parachute. This bug is reported to ArduPilot developers, and we are waiting for a reply from them.

2) Case Study 2 - Failing to Maintain Proper Altitude after the Flip Maneuver: Each configuration parameter has its valid range. However, PGFUZZ discovered that a set of parameters have incorrect valid ranges, which causes the vehicle to crash on the ground.

Policy. The ArduPilot documentation states that if the throttle stick is in the middle position (i.e., maintaining the current altitude), and the vehicle is in `ALT_HOLD` mode, it must maintain the current altitude. We represent this requirement with $\texttt{A.ALT\_HOLD}_2$ policy (See Table XII for its MTL).

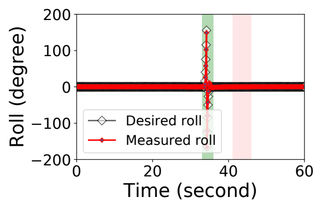
Root Cause. PGFUZZ discovered that this requirement is not correctly implemented if roll axis rate controller parameters are changed. Figure 10b shows the drone's altitude decreases when the vehicle is in the `FLIP` mode (second 22) under normal conditions. ArduPilot is able to maintain the current altitude when switching back to the `ALT_HOLD` mode (starting from second 24). However, if the values of the two roll axis rate controller parameters (`ATC_RATE_RLL_FF` and `ATC_RATE_R_MAX`) are modified, ArduPilot cannot maintain the altitude after a flip maneuver although the flight mode is the `ALT_HOLD` and the throttle stick is in the middle, as shown in Figure 10c. The root cause is the broad range of accepted parameter values.

Attack. An attacker can exploit this vulnerability by assigning a small value to the two roll axis rate controller parameters. Here the attacker can manipulate the configuration parameters by either overriding them before a flight or changing them after the vehicle takes off. When the user triggers the `FLIP` mode, the drone fails to recover a stable roll angle due to the limited roll angular velocity, which leads to the failure to stay at the same altitude and eventually crashing to the ground. We note that the changed roll parameters do not affect the drone's attitude control under its normal operation since they do not require a large roll angle velocity. This prevents users from noticing the limited roll angle velocity and, consequently, the attack. To prevent such unstable attitude control, ArduPilot requires to increase the minimum range values of `ATC_RATE_R_-MAX` and `ATC_RATE_RLL_FF`. This bug is confirmed by ArduPilot developers, and they proposed to update the parameter ranges.

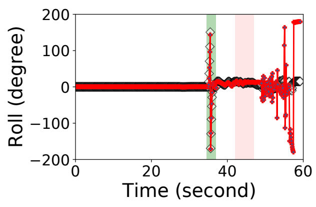
3) Case Study 3 - Incorrect Altitude Computation after Acrobatic Flying: Drones measure the same physical state from multiple sensors to address sensor failures and perform sensor fusion. For instance, GPS and barometer sensors measure altitude simultaneously. However, PGFUZZ discovered that ArduPilot incorrectly computes the altitude when high deviations in GPS sensor occur. The drone cannot maintain its altitude in the `ALT_HOLD` mode due to the incorrect altitude.

Policy. PGFUZZ discovered this policy violation while fuzzing the $\texttt{A.ALT\_HOLD}_2$ policy, which is the same policy discussed in Case Study 2 (Section VII-C2). While the violated policies are the same, the input sequence that causes the violation and the violation's root cause differs. This is because PGFUZZ first reboots the drone on the simulator to negate all changed configurations after finding a policy violation. It then restarts fuzzing to find different bugs related to the same policy.

Root Cause. PGFUZZ discovered that high deviations in GPS sensor coupled with incorrectly assigned parameter values result in the drone not maintaining its altitude in the `ALT_HOLD` mode. First, PGFUZZ causes a high deviation in GPS sensor measurements by assigning a value to the `GPS_POS1_Z` parameter and triggering an acrobatic flying activity (e.g., `FLIP` or `ACRO` modes). The high deviation causes ArduPilot to switch the altitude measurement source from GPS to the barometer. Then, ArduPilot incorrectly applies the `GND_ALT_OFFSET` parameter to calculate the barometric altitude, causing an undesired altitude change, as shown in Figure 10d. Particularly, ArduPilot sets the altitude to zero when the vehicle is taking off, although the user assigns a value to the `GND_ALT_OFFSET` before take-off. Hence, ArduPilot



(a) Roll angles under normal conditions. It shows that the drone safely lands on the ground.



(b) Roll angles under the attack triggering Figure 10d and changes the `LAND_SPEED_HIGH` parameter.

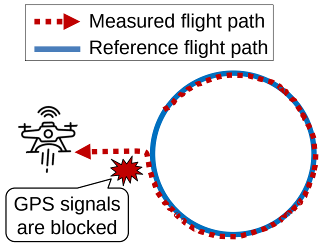
Fig. 11: Landing on the ground in different scenarios. The green and red colors have the same meaning with Figure 10.

must not apply the offset to calculate the altitude after take-off. However, PGFuzz found that ArduPilot uses the offset, which prevents the vehicle from staying at the same altitude.

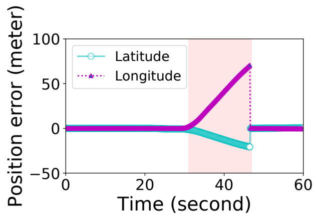
Attack. An attacker can exploit this policy violation and crash the drone, although the violation itself does not cause a physical crash. Particularly, the attacker (1) configures the `FS_EKF_ACTION` parameter to land the drone on the ground when the measured GPS values deviate and (2) assigns a large value to the `GND_ALT_OFFSET` and `LAND_SPEED_HIGH` parameters. Then, when a user executes an acrobatic flying activity (e.g., `FLIP` mode), it triggers the policy violation (Figure 10d). When the user executes the acrobatic flying activity, GPS sensor values deviate. Hence, the drone starts landing due to the attacker's configuration on the `FS_EKF_ACTION` parameter. While the drone is landing on the ground, it uses two different descent speeds: (1) `LAND_SPEED_HIGH` parameter is the descent speed when the drone is higher than 10 meters from the ground and (2) `LAND_SPEED` parameter is the descent speed when the drone's altitude is less than 10 meters. During the attack, the drone keeps using the `LAND_SPEED_HIGH` while landing instead of the `LAND_SPEED` because of the miscalculated altitude. In other words, the drone misjudges the current altitude and maintains a fast descent speed even though the altitude is less than 10 meters. Figure 11a illustrates the drone's safe landing under normal conditions, whereas Figure 11b depicts that the drone hits the ground with 12.86 m/s when it is under attack. The attacker can stealthily conduct this attack because (1) the parameter changes do not cause any noticeable difference in the drone's normal operation, and (2) after the attacker changes these parameters, the bug is triggered when the user executes the acrobatic flying. To prevent this bug, ArduPilot should check all of the altitude values to stay at the same altitude while the RV is in the `ALT_HOLD` mode instead of only checking the previous altitude. We reported the bug to ArduPilot developers. However, we are still awaiting a reply from them.

4) Case Study 4 - Failing to Trigger the GPS Fail-safe: PGFuzz discovered that assigning a negative value to the `COM_POS_FS_DELAY` parameter, which represents the time delay in turning on a GPS fail-safe and setting to specific flying modes cause PX4 to fail to trigger the GPS fail-safe.

Policy. PX4 documentation states that if time exceeds `COM_POS_FS_DELAY` seconds after GPS loss is detected, the GPS fail-safe must be triggered. We express this requirement with $\mathtt{PX.GPS.FS}_1$ policy (See Table XII for its MTL). We note that the time constraint of the MTL formula does not include a constant upper bound (i.e., $\Diamond_{[0,\mathtt{COM\_POS\_FS\_DELAY}+\mathtt{k}]}$) but depends on a variable k. This is because triggering the GPS fail-safe requires `COM_POS_FS_DELAY` time and an additional time delay (k). The additional delay k is caused by the soft real-time system's task scheduling. We repeatedly measured the additional time delay k and noticed its maximum value is less than a second. However, to conservatively detect a policy violation, we set the upper bound to the `COM_POS_FS_DELAY` twice of the maximum delay time.



(a) PX4 maintains the `ORBIT` flight mode under GPS signal loss.



(b) PX4 lands on the ground if the drone navigates to a location under GPS signal loss.

Fig. 12: Illustration of failing to trigger the GPS fail-safe in different scenarios. The red color box represents when the GPS signals are blocked (30-47 secs).

Root Cause. The violation happens because PX4 developers do not implement a parameter range check. PX4 v1.7.4 forces `COM_POS_FS_DELAY` parameter to have a value in the valid range. Thereafter, it checks whether the GPS fail-safe needs to be triggered. However, we found that the code lines to check the `COM_POS_FS_DELAY` parameter are removed by developers in PX4 v1.9 while updating the fail-safe code snippets. When a user assigns a negative value to the parameter, it affects the decision to trigger the fail-safe when the current flight mode is `ORBIT` or the drone is flying into a location. Specifically, if the flight mode is not `ORBIT` or the drone stays at the same location, PX4 correctly triggers the GPS fail-safe. This observation makes it difficult for the developers to notice the bug.

PGFuzz uncovered the bug by assigning a negative value to the `COM_POS_FS_DELAY` parameter, changing the current flight mode to `ORBIT` and turning off GPS signals. As a result, the drone stopped the navigation and aimed at staying in the current location via inertial measurement unit (IMU) sensors (e.g., accelerometers) instead of turning on the GPS fail-safe. However, accumulated errors from the IMU caused the drone to randomly float in the air depending on the wind directions, as shown in Figure 12a.

Attack. An attacker can exploit this policy violation to prevent the drone from handling scenarios in which the GPS signal is lost, eventually leading to a physical crash of the drone. Specifically, the attacker can assign a negative value to the `COM_POS_FS_DELAY` parameter to trigger this bug. When the drone passes through an area where the GPS signal is not available, the drone will fail to turn on the proper GPS fail-safe, and it will fly to a wrong location, as shown in Figure 12a and Figure 12b. In our example, the vehicle deviates from its planned route by up to 20.7 meters in its latitude and 70.5 meters in its longitude. Such a deviated flight path could potentially make the vehicle physically crash.

The described attack can be stealthily performed, since the loss of GPS signal is normal behavior that occurs naturally, especially in some circumstances, such as when the drone flies in highly urbanized areas. To prevent this bug, PX4 should restore the previous valid range check statement. We reported the bug to PX4 developers, and they accepted it.

	Types of bugs			
Framework	**Software crash**	**Control instability**	**Mis-implementation**	**Unimplemented command**
Conventional Fuzzing Methods [5], [44], [48]	✓	✗	✗	✗
AR-SI [36]	✗	✓	✗	✗
RVFuzzer [41]	✗	✓	✗	✗
MAYDAY [40]	✓	✓	✗	✗
Control Invariant [24]	✗	✓	✗	✗
Cyber-Physical inconsistency [23]	✗	✓	✓	✗
Fuzzing for CPSs [22]	✗	✓	✗	✗
PGFuzz	✓	✓	✓	✓

TABLE X: A comparison of PGFUZZ with other fuzzing works.

VIII. RELATED WORK

In comparison to other approaches including traditional fuzzing methods [5], [44], [48], CPS fuzzing works [21], [22], and control instability detection works [23], [24], [36], [40], [41] as presented in Table X, PGFUZZ tests the whole control software by allowing users to define any functional requirements in the form of MTL formulas. This enables PGFUZZ to discover additional types of bugs such as checking for safety conditions (e.g., when a drone opens the parachute, as explained in Section VII-C1), drone physical crashes due to parameters' incorrect ranges (Section VII-C2), and incorrect altitude calculation (Section VII-C3). We note that such bugs can only be discovered by PGFUZZ.

Traditional fuzzing methods [5], [44], [48] mainly discovers memory corruption vulnerabilities (e.g., buffer overflow). However, PGFUZZ can discover new types of bugs, including misimplementation, control instability, and unimplemented commands leading to undesired vehicle states.

Chen et al. [21] aims to detect triggered bugs by checking if sensor or actuator values go outside specific safe bounds. Unfortunately, this assumption is not always true in the scenarios we consider in this paper. In fact, it is possible that a drone is in a state in which every sensor reports reasonable values, while, however, being in an unsafe state. For instance, consider a drone having a vertical speed of 3 m/s. While this value is acceptable under normal operations, it is unsafe during, for instance, the `ALT_HOLD` flight mode. To detect this kind of violations, PGFUZZ uses MTL policies to capture temporal and causal relations among states of RVs. This allows PGFUZZ to identify a larger range of bugs.

In addition, there exist approaches designed to find bugs specifically in RVs [23], [24], [36], [40], [41]. Specifically, RVFuzzer [41], as detailed in Section VII-B, fuzzes the configuration parameters and some environment factors (e.g., wind) to find input validation bugs. MAYDAY [40] localizes bugs in the source code which lead to unstable attitude/crash. AR-SI [36] uses autoregressive system identification to detect control instability bugs. Choi et al. [24] extracts a control invariant model representing the robotic vehicle's dynamics and control algorithm. The model takes the RV's states and predicts the next states related to attitude control (i.e., roll, pitch, and yaw angles). If a substantial state difference is measured between an RV and the model, it assumes that an undesired state change has occurred. These approaches detect bugs that cause either unstable attitudes or flight path changes. For this reason, they are unable to find other types of bugs, outside of these two categories, such as the bug explained in Section VII-C4. On the contrary, PGFUZZ can discover a larger variety of bugs affecting RVs, since it can detect violations of any property that can be expressed with an appropriate MTL formula.

Choi et al. [23] also proposed a technique to find bugs in safety checks of drones. This approach mutates environment conditions (e.g., wind and mass of physical objects) to verify whether code snippets perform sanity checks for detecting a set of safety-critical cases. They mainly use the model derived in their previous work [24] as a bug oracle. In contrast, PGFUZZ tests the whole control software by allowing users to define any functional requirements in the form of MTL formulas. This enables PGFUZZ to discover additional types of bugs such as the lack of checking for safety conditions (when, for instance, a drone opens the parachute, as explained in Section VII-C1), drone physical crashes due to parameters' incorrect ranges (Section VII-C2), and incorrect altitude calculation (Section VII-C3).

Formal methods are also used to discover bugs in RVs [30], [31]. However, their models often suffer from state explosion problems, which limits them from porting to complex systems such as RVs. There have also been efforts to build formal verification to detect safety issues via machine learning techniques [2], [3]. However, they focus on malicious sensor/actuator faults and spoofing attacks instead of RV software bugs.

IX. LIMITATIONS AND DISCUSSION

Imperfection of RV Simulators. We use Software-in-the-Loop (SITL) as our testing environment. Imperfect simulations could cause two issues. First, if simulators incorrectly simulate the vehicle's states and/or hardware, PGFUZZ will identify false-positive policy violations. However, we confirmed that all policy violations found by PGFUZZ could be reproduced on a real vehicle. We used a 3DR IRIS + UAV platform equipped with the Pixhawk 1 flight management unit board in our experiments. Second, if the simulators do not support specific hardware (e.g., RFD 900 radio modem [14]), PGFUZZ cannot find bugs in those hardware modules. To address this, PGFUZZ can be integrated into Hardware-in-the-loop (HIL) simulation [11], [55] or Simulation-In-Hardware (SIH) [54] where firmware is run on real flight controller hardware. Yet, the HIL and SIH require numerous hardware devices to vet all hardware configurations.

Monitoring Real-time Properties of Temporal Logic. Since PGFUZZ checks policies at run-time during a simulation, at time point $\mathtt{t}$, only the data traces for $\mathtt{1}$,...,$\mathtt{t}$ are available to check the policies. Therefore, MTL policies with unbounded future operators cannot be checked at run-time. Following the online monitoring systems [16], [27], we define the policies with a subclass of MTL that considers unbounded past and bounded future. To illustrate, consider a policy that states the altitude of a vehicle must eventually exceed 10 meters, defined as $\Box(\mathtt{ALT} > 10)$. This policy cannot be checked at time $\mathtt{t}$ since it depends on the drone's future states not yet available to PGFUZZ. However, when this policy is restricted with a bounded future such as $\Box_{[0,5]}(\mathtt{ALT} > 10)$ (the altitude must eventually exceed 10 meters within 5 seconds), the policy can be checked at time $\mathtt{t}+5$.

Porting PGFUZZ to other RVs. Users can port PGFUZZ to other types of RV software by following six steps. These steps are required for policy identification, finding the maximum number of bugs, and minimizing the fuzzing time: (1) create MTL (or LTL) policies for the RV, (2) identify new states that are not included in the states list defined by PGFUZZ, (3) update the synonym table (See Figure 5), (4) map MTL formula terms to variables in the source code (See Figure 4), (5) verify and update policy violation predicates according to the new MTL formulas (See Figure 13), and (6) exclude self-sabotaging inputs leading to false-positive policy violations (See Appendix C). The porting effort depends on how similar the two platforms are. We believe that such a workload is not a significant burden for developers or knowledgeable users. For example, when porting PGFUZZ from ArduPilot to PX4, the total porting effort took 6.3 hours. This includes modifying 54 LoC in the Pre-Processing and 94 LoC in the mutation engine to adapt to differences in MAVLink protocol. The required time for the manual porting effort is detailed in Appendix D.

X. CONCLUSIONS

We introduce PGFUZZ, a policy-guided fuzzing framework, which leverages policies represented by temporal logic with timing constraints to find bugs in robotic vehicle control software. PGFUZZ addresses the unique challenges in fuzzing RVs by (1) reducing the large input space through static and dynamic analysis, and (2) mutating fuzzing inputs to minimize a distance metric that measures "how close" the RV's current state is to a policy violation. We evaluated PGFUZZ on three popular flight control software packages, ArduPilot, PX4, and Paparazzi. PGFUZZ discovered 156 previously unknown bugs, and 128 of the bugs can only be discovered with PGFUZZ. We reported the bugs to the software developers of flight control software, and they acknowledged 106 of these bugs. Future work will expand our analysis to support more safety-critical systems and study the safety and security requirements engineering process to discover more complex policies.

ACKNOWLEDGMENT

This work was supported in part by ONR under Grants N00014-20-1-2128 and N00014-17-1-2045. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the ONR.

REFERENCES

[1] H. Abbas, B. Hoxha, G. Fainekos, and K. Ueda, "Robustness-guided temporal logic testing and verification for stochastic cyber-physical systems," in *IEEE International Conference on Cyber Technology in Automation, Control and Intelligent*, 2014.

[2] A. Abbaspour, P. Aboutalebi, K. K. Yen, and A. Sargolzaei, "Neural adaptive observer-based sensor and actuator fault detection in nonlinear systems: Application in uav," *ISA transactions*, 2017.

[3] A. Abbaspour, K. K. Yen, S. Noei, and A. Sargolzaei, "Detection of fault data injection attack on uav using adaptive neural network," *Procedia computer science*, 2016.

[4] M. Abrams and J. Weiss, "Malicious control system cyber security attack case study–maroochy water services, australia," *McLean, VA: The MITRE Corporation*, 2008.

[5] "Afl," http://lcamtuf.coredump.cx/afl, 2020.

[6] C. M. Ahmed, M. Ochoa, J. Zhou, A. P. Mathur, R. Qadeer, C. Murguia, and J. Ruths, "Noiseprint: Attack detection using sensor and process noise fingerprint in cyber physical systems," in *Asia Conference on Computer and Communications Security*, 2018.

[7] "Amazon prime air," https://tinyurl.com/hlb4e22, 2020.

[8] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, University of Cophenhagen, 1994.

[9] "Apm sitl," https://tinyurl.com/yxespupu, 2020.

[10] "Ardupilot," https://ardupilot.org/, 2020.

[11] "Ardupilot hardware in the loop," https://tinyurl.com/yxemuknb, 2020.

[12] "Ardupilot parameter list," https://tinyurl.com/y3g6gvgx, 2020.

[13] "Ardupilot parachute," https://tinyurl.com/yxgf23yj, 2020.

[14] "Ardupilot rfd900 radio modem," https://tinyurl.com/y4rbjun6, 2020.

[15] "Ardupilot source code," https://github.com/ArduPilot/ardupilot, 2020.

[16] D. Basin, B. N. Bhatt, and D. Traytel, "Almost event-rate independent monitoring of metric temporal logic," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2017.

[17] "Boeing 737 max crash: Anti-stall system reportedly activated in ethiopian airlines tragedy," https://tinyurl.com/y476slxf, 2019.

[18] J. build team, "Jsbsim," https://github.com/JSBSim-Team/jsbsim, 2020.

[19] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated IoT safety and security analysis," in *USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[20] P. Chen, M. Dean, D. Ojoko-Adams, H. Osman, and L. Lopez, "Systems quality requirements engineering (square) methodology: Case study on asset management system," CMU Software Engineering institute, Tech. Rep., 2004.

[21] Y. Chen, C. M. Poskitt, and J. Sun, "Learning from mutants: Using code mutation to learn and monitor invariants of a cyber-physical system," in *IEEE Symposium on Security and Privacy (SP)*, 2018.

[22] Y. Chen, C. M. Poskitt, J. Sun, S. Adepu, and F. Zhang, "Learning-guided network fuzzing for testing cyber-physical system defences," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[23] H. Choi, S. Kate, Y. Aafer, X. Zhang, , and D. Xu, "Cyber-physical inconsistency vulnerability identification for safety checks in robotic vehicles," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[24] 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 *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[25] D. R. Clark, C. Meffert, I. Baggili, and F. Breitinger, "Drop (drone open source parser) your drone: Forensic analysis of the dji phantom iii," *Digital Investigation*, vol. 22, 2017.

[26] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on Logic of Programs*, 1981.

[27] A. Dokhanchi, B. Hoxha, and G. Fainekos, "On-line monitoring for temporal logic robustness," in *International Conference on Runtime Verification*, 2014.

[28] "Exec shield," https://tinyurl.com/y5nk9egy, 2005.

[29] G. Fainekos, B. Hoxha, and S. Sankaranarayanan, "Robustness of specifications and its applications to falsification, parameter mining, and runtime monitoring with s-taliro," in *International Conference on Runtime Verification*, 2019.

[30] C. Fan, B. Qi, S. Mitra, and M. Viswanathan, "Dryvr: data-driven verification and compositional reasoning for automotive systems," in *International Conference on Computer Aided Verification*, 2017.

[31] C. Fan, B. Qi, S. Mitra, M. Viswanathan, and P. S. Duggirala, "Automatic reachability analysis for nonlinear hybrid models with c2e2," in *International Conference on Computer Aided Verification*, 2016.

[32] B. Gati, "Open source autopilot for academic research-the paparazzi system," in *American Control Conference*, 2013.

[33] "Gazebo," http://gazebosim.org, 2020.

[34] A. Ginter, "The top 20 cyberattacks on industrial control systems," https://tinyurl.com/y3honz7s, 2020.

[35] "Google x-wing," https://x.company/projects/wing, 2020.

[36] Z. He, Y. Chen, E. Huang, Q. Wang, Y. Pei, and H. Yuan, "A system identification based oracle for control-cps software fault localization," in *IEEE/ACM International Conference on Software Engineering*, 2019.

[37] K. Jansen, M. Schäfer, D. Moser, V. Lenders, C. Pöpper, and J. Schmitt, "Crowd-gps-sec: Leveraging crowdsourcing to detect and localize gps spoofing attacks," in *IEEE Symposium on Security and Privacy (SP)*, 2018.

[38] M. Jo, J. Park, Y. Baek, R. Ivanov, J. Weimer, S. H. Son, and I. Lee, "Adaptive transient fault model for sensor attack detection," in *IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, 2016.

[39] C. H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, and D. Xu, "Securing real-time microcontroller systems through customized memory view switching." in *NDSS*, 2018.

[40] 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 *USENIX Security*, 2020.

[41] 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 *USENIX Security*, 2019.

[42] R. Koymans, “Specifying real-time properties with metric temporal logic,” *Real-time systems*, vol. 2, no. 4, 1990.

[43] Y.-M. Kwon, J. Yu, B.-M. Cho, Y. Eun, and K.-J. Park, “Empirical analysis of mavlink protocol vulnerability for attacking unmanned aerial vehicles,” *IEEE Access*, 2018.

[44] “libfuzzer,” https://llvm.org/docs/LibFuzzer.html, 2020.

[45] “Llvm,” https://tinyurl.com/y5hejsra, 2020.

[46] “Log analyzer,” https://tinyurl.com/y29ncej9, 2020.

[47] “Mavlink,” https://mavlink.io/en, 2020.

[48] “Oss-fuzz,” https://google.github.io/oss-fuzz/, 2020.

[49] “Paparazzi dev team nps,” https://wiki.paparazziuav.org/wiki/NPS, 2020.

[50] A. Pnueli, “The temporal logic of programs,” in *Annual Symposium on Foundations of Computer Science*, 1977.

[51] “Pprzlink,” https://github.com/paparazzi/pprzlink, 2020.

[52] “Px4 open source drone,” https://px4.io/, 2020.

[53] “Px4 parameter list,” https://tinyurl.com/y2tng477, 2020.

[54] “Px4 simulation-in-hardware,” https://tinyurl.com/y5rv7cpd, 2020.

[55] “Px4 hardware in the loop,” https://tinyurl.com/y5gfwosw, 2020.

[56] “Pymavlink,” https://github.com/ArduPilot/pymavlink, 2020.

[57] “Pyparsing,” https://github.com/pyparsing/pyparsing, 2020.

[58] N. M. Rodday, R. d. O. Schmidt, and A. Pras, “Exploring security vulnerabilities of unmanned aerial vehicles,” in *IEEE/IFIP Network Operations and Management Symposium*, 2016.

[59] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Global value numbers and redundant computations,” in *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1988.

[60] “Smart traffic signals,” https://tinyurl.com/yxvyntpb, 2020.

[61] Y. Son, H. Shin, D. Kim, Y. Park, J. Noh, K. Choi, J. Choi, and Y. Kim, “Rocking drones with intentional sound noise on gyroscopic sensors,” in *USENIX Security*, 2015.

[62] Y. Sui and J. Xue, “Svf: interprocedural static value-flow analysis in llvm,” in *International conference on compiler construction*, 2016.

[63] “Tesla crash,” https://tinyurl.com/y3sws2wr, 2016.

[64] H. Yang, G. Fainekos, H. Sarjoughian, and A. Shrivastava, “Dynamic programming algorithm for computing temporal logic robustness,” Ph.D. dissertation, Arizona State University, 2013.

[65] 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 *USENIX Security*, 2018.

[66] L. Zhang, W. He, J. Martinez, N. Brackenbury, S. Lu, and B. Ur, “Autotap: synthesizing and repairing trigger-action programs using ltl properties,” in *IEEE/ACM International Conference on Software Engineering*, 2019.

APPENDIX

A. List of Terms

Table XI shows the physical states extracted from policies presented in Table XII. We also include a list of $\mathtt{Input_P}$ and $\mathtt{Input_E}$ in the list of terms.

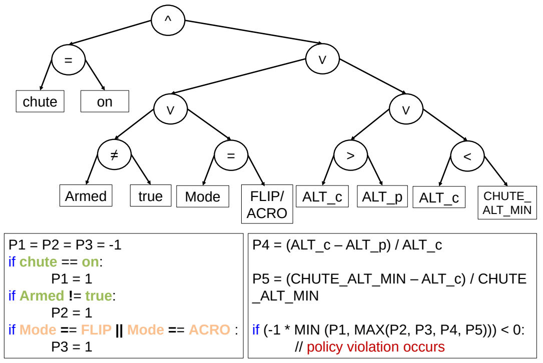
B. Automated Predicate Generation

Users who desire to port PGFUZZ in other RV software need to conduct six steps to find the maximum number of bugs and minimize the fuzzing time as explained Section IX (Porting PGFUZZ to other RVs).

To reduce users’ manual effort, we automate the fifth step (i.e., creating code snippets to calculate propositional and global distances). We call this step predicate generator. The predicate generator first parses and verifies MTL formulas based on the BNF definition in Section V-A1. We implemented the analyzer via PyParsing library [57]. Second, to arithmetically calculate the distances in the source code, it converts the verified MTL formula in the *always* form into an MTL formula in the *not eventually* form as explained in Section V-B3. Third, we create the binary expression tree based on the converted MTL formula (as shown in Figure 13). Then, we traverse the nodes of the tree to automatically generate the code snippets that calculate the distances and check a policy violation. We created the code snippets of propositional and global distances, as explained in Section V-B3.

ID	Type	State	Description
S_1	Position	latitude, longitude, altitude	(x,y,z) location of the vehicle
S_2	Attitude	roll, pitch, yaw, roll speed, pitch speed, yaw speed, reference roll, reference pitch, reference yaw	Measured and desired attitude
S_3	Operation	air speed, ground speed, throttle, climb rate, reference air speed, flight mode, parachute	Physical movement and operation mode
S_4	RC inputs	RC 1 - 4	Radio channel inputs from users
S_5	System	system clock, flight status, mission, pre-arm checking	System general info. such as the vehicle is on the ground or free falling
S_6	Sensor	gyroscope, accelerometer, magnetometer, barometer, GPS	Sensor condition

TABLE XI: The identified physical states. S_1-S_5 are obtained from MAVLink and S_6 is by parsing the ACK messages from the vehicle.



```
P1 = P2 = P3 = -1
if chute == on:
        P1 = 1
if Armed != true:
        P2 = 1
if Mode == FLIP || Mode == ACRO :
        P3 = 1
```

```
P4 = (ALT_c – ALT_p) / ALT_c

P5 = (CHUTE_ALT_MIN – ALT_c) / CHUTE
_ALT_MIN

if (-1 * MIN (P1, MAX(P2, P3, P4, P5))) < 0:
        // policy violation occurs
```

Fig. 13: Illustration of $\mathtt{A.CHUTE_1}$ policy’s binary expression tree. c and p denote current time (t) and previous time (t − 1). A python code represents propositional distances (P1 − P5) and a global distance derived by P1 − P5.

C. Handling False Positives

We exclude the following $\mathtt{Input_{set}}$ which leads to false positive policy violations: (1) $\mathtt{Input_P}$ influencing hardware configurations (e.g., device IDs), and (2) $\mathtt{Input_C}$ to turn off engines.

D. Required Time for Porting Effort

When users port PGFUZZ to other RV software, some manual tasks are required as presented in Section IX (Porting PGFUZZ to other RVs). We spent a total of 23.4 hours on the manual effort. Specifically, we deployed PGFUZZ in the order of ArduPilot, PX4, and Paparazzi, and the manual effort took 13.5, 6.3, and 3.6 hours, respectively. We spent less time on PX4 and Paparazzi than ArduPilot because the flight control programs have similar or same flight modes, synonyms, and self-sabotaging inputs.

E. Policy Descriptions

To evaluate PGFUZZ, we use the following 56 policies, formally expressed with MTL formulas in Table XII.

ID	Policy Description	Templ.	MTL notation
A.RTL$_1$	If the current altitude is less than `RTL_ALT`, then altitude must be increased until the altitude is greater or equal to the `RTL_ALT`.	T$_3$	$\square\{(\mathtt{ALT}_t < \mathtt{RTL\_ALT}) \land (\mathtt{Mode}_t = \mathtt{RTL}) \rightarrow (\mathtt{ALT}_{t-1} < \mathtt{ALT}_t)\}$
A.RTL$_2$	If the current altitude is greater or equal to `RTL_ALT`, current flight mode is `RTL`, and the current vehicle is not at `home position`, then the vehicle must move to the `home position` while maintaining the current altitude.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{RTL}) \land (\mathtt{ALT}_t \geq \mathtt{RTL\_ALT}) \land (\mathtt{Pos}_t \neq \mathit{home\,position}) \rightarrow (\mathtt{Pos}_{t-1} \neq \mathtt{Pos}_t) \land (\mathtt{ALT}_{t-1} = \mathtt{ALT}_t)\}$
A.RTL$_3$	If current altitude is greater or equal to `RTL_ALT` and current position is the same as `home position`, then flight mode must be `LAND`.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{RTL}) \land (\mathtt{ALT}_t \geq \mathtt{RTL\_ALT}) \land (\mathtt{Pos}_t = \mathit{home\,position}) \rightarrow (\mathtt{Mode}_t = \mathtt{LAND})\}$
A.RTL$_4$	If current flight mode is `LAND` and the vehicle touches the ground, then the vehicle must disarm motors.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{LAND}) \land (\mathtt{ALT}_t = \mathtt{Ground}_{\mathtt{ALT}}) \rightarrow (\mathtt{Disarm} = \mathtt{on})\}$
A.FLIP$_1$	If and only if roll is less than 45 degree, throttle is greater or equal to 1,500, altitude is more than 10 meters, and the current flight mode is one of `ACRO` and `ALT_HOLD`, then the flight mode can be changed to `FLIP`.	T$_2$	$\square\{(\mathtt{Mode}_t = \mathtt{FLIP}) \rightarrow (\mathtt{Mode}_{t-1} = \mathtt{ACRO/ALT\_HOLD}) \land \neg(\mathtt{Roll}_t > 45) \lor (\mathtt{Throttle} \leq 1{,}500) \lor (\mathtt{ALT}_t < 10)\}$
A.FLIP$_2$	If the current flight mode is `FLIP` and roll is between -90 and 45 degree, then rolling right at 400 degree per second.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{FLIP}) \land (-90 \leq \mathtt{Roll}_t \leq 45) \rightarrow (\mathtt{Roll}_{\mathtt{rate}} = 400) \land (\mathtt{Roll}_{\mathtt{Direction}} = \mathtt{right})\}$
A.FLIP$_3$	After the vehicle finishes A.FLIP$_2$, the vehicle must recover the original attitude (i.e., roll, pitch, and yaw) within k seconds.	T$_1$ & T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{FLIP}_3) \rightarrow (\mathtt{Roll}_t = \Diamond_{[0,K]}\mathtt{Roll}_{\mathtt{Original}}) \land (\mathtt{Pitch}_t = \Diamond_{[0,K]}\mathtt{Pitch}_{\mathtt{Original}}) \land (\mathtt{Yaw}_t = \Diamond_{[0,K]}\mathtt{Yaw}_{\mathtt{Original}})\}$
A.FLIP$_{\mathrm{General}}$	The vehicle should complete the rolling (A.FLIP$_2$) within 2.5 seconds and must return to the original flight mode.	T$_1$	$\square\{(\mathtt{Mode}_t = \mathtt{FLIP}_1) \rightarrow (\Diamond_{[0,2.5]}\mathtt{Mode}_t = \mathtt{FLIP}_3)\}$
A.ALT_HOLD$_1$	If the altitude source is the barometer, the vehicle must follow the altitude computed by this source, rather than the GPS.	T$_3$	$\square\{(\mathtt{ALT}_{\mathtt{src}} = \mathit{Baro}) \rightarrow (\mathtt{ALT}_t = \mathtt{ALT}_{\mathtt{Baro}}) \land (\mathtt{ALT}_t \neq \mathtt{ALT}_{\mathtt{GPS}})\}$
A.ALT_HOLD$_2$	If the throttle stick is in the middle (i.e., 1,500) the vehicle must maintain the current altitude.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{ALT\_HOLD}) \land (\mathtt{Throttle}_t = 1{,}500) \rightarrow (\mathtt{ALT}_t = \mathtt{ALT}_{t-1})\}$
A.CIRCLE$_1$	Pitch stick up must reduce the radius until it reaches zero.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{CIRCLE}) \land (\mathtt{RC}_{\mathtt{pitch}} < 1{,}500) \land (\mathtt{Circle\_radius}_t > 0) \rightarrow (\mathtt{Circle\_radius}_t < \mathtt{Circle\_radius}_{t-1})\}$
A.CIRCLE$_2$	Pitch stick down must increase the radius.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{CIRCLE}) \land (\mathtt{RC}_{\mathtt{pitch}} > 1{,}500) \rightarrow (\mathtt{Circle\_radius}_t > \mathtt{Circle\_radius}_{t-1})\}$
A.CIRCLE$_3$	Roll stick right (think clockwise) must increase the speed while moving clockwise.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{CIRCLE}) \land (\mathtt{RC}_{\mathtt{roll}} > 1{,}500) \land (\mathtt{Circle\_direction}_t = \mathit{clockwise}) \rightarrow (\mathtt{Circle\_speed}_t > \mathtt{Circle\_speed}_{t-1})\}$
A.CIRCLE$_4$	Roll stick right (think clockwise) must decrease the speed while moving counterclockwise.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{CIRCLE}) \land (\mathtt{RC}_{\mathtt{roll}} > 1{,}500) \land (\mathtt{Circle\_direction}_t = \mathit{counterclockwise}) \rightarrow (\mathtt{Circle\_speed}_t < \mathtt{Circle\_speed}_{t-1})\}$
A.CIRCLE$_5$	Roll stick left (think counterclockwise) must increase the speed while moving counterclockwise.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{CIRCLE}) \land (\mathtt{RC}_{\mathtt{roll}} < 1{,}500) \land (\mathtt{Circle\_direction}_t = \mathit{counterclockwise}) \rightarrow (\mathtt{Circle\_speed}_t > \mathtt{Circle\_speed}_{t-1})\}$
A.CIRCLE$_6$	Roll stick left (think counterclockwise) must decrease the speed while moving clockwise.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{CIRCLE}) \land (\mathtt{RC}_{\mathtt{roll}} < 1{,}500) \land (\mathtt{Circle\_direction}_t = \mathit{clockwise}) \rightarrow (\mathtt{Circle\_speed}_t < \mathtt{Circle\_speed}_{t-1})\}$
A.CIRCLE$_7$	The users do not have any control over the roll, pitch, and yaw but can change the altitude with the throttle stick.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{CIRCLE}) \rightarrow (\mathtt{RC\_roll}_t/\mathtt{RC\_pitch}_t/\mathtt{RC\_yaw}_t = \mathtt{RC\_roll}_{t-1}/\mathtt{RC\_pitch}_{t-1}/\mathtt{RC\_yaw}_{t-1}) \land \{(\mathtt{RC\_throttle}_t \leq \mathtt{RC\_throttle}_{t-1}) \lor (\mathtt{RC\_throttle}_t \geq \mathtt{RC\_throttle}_{t-1})\}\}$
A.LAND$_1$	Above 10 meters the vehicle must descend at the rate specified in the `LAND_SPEED_HIGH` parameter	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{LAND}) \land (\mathtt{ALT}_t \geq 10) \land \rightarrow (\mathtt{Speed\_vertical}_t = \mathtt{LAND\_SPEED\_HIGH})\}$
A.LAND$_2$	Below 10 meters the vehicle must descend at the rate specified in the `LAND_SPEED` parameter.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{LAND}) \land (\mathtt{ALT}_t < 10) \land \rightarrow (\mathtt{Speed\_vertical}_t = \mathtt{LAND\_SPEED})\}$
A.AUTO$_1$	The pilot's roll, pitch and throttle inputs must be ignored but the yaw can be overridden with the yaw stick.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{AUTO}) \rightarrow (\mathtt{RC\_roll}_t/\mathtt{RC\_pitch}_t/\mathtt{RC\_throttle}_t = \mathtt{RC\_roll}_{t-1}/\mathtt{RC\_pitch}_{t-1}/\mathtt{RC\_throttle}_{t-1}) \land \{(\mathtt{RC\_yaw}_t \leq \mathtt{RC\_yaw}_{t-1}) \lor (\mathtt{RC\_yaw}_t \geq \mathtt{RC\_yaw}_{t-1})\}\}$
A.BRAKE$_1$	When the vehicle is in `BRAKE` mode, it must stop within `k` seconds	T$_1$	$\square\{(\mathtt{Mode}_t = \mathtt{BRAKE}) \rightarrow (\Diamond_{[0,k]}\mathtt{Pos}_t = \mathtt{Pos}_{t-1})\}$
A.DRIFT$_1$	If the vehicle loses GPS signals in flight while in `DRIFT` mode, the vehicle must either `LAND` or enter `ALT_HOLD` mode based on `FS_EKF_ACTION` parameter.	T$_1$	$\square\{(\mathtt{GPS}_{\mathtt{fail}} = \mathit{on}) \land (\mathtt{Mode}_t = \mathtt{DRIFT}) \rightarrow (\Diamond_{[0,k]}\mathtt{Mode}_t = \mathtt{FS\_EKF\_ACTION})\}$
A.LOITER$_1$	The vehicle must maintain a constant location, heading, and altitude.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{LOITER}) \rightarrow (\mathtt{Pos}_t = \mathtt{Pos}_{t-1} \land \mathtt{Yaw}_t = \mathtt{Yaw}_{t-1} \land \mathtt{ALT}_t = \mathtt{ALT}_{t-1})\}$
A.GUIDED$_1$	If there is no more way point, the vehicle must stay at the same location, heading, and altitude.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{GUIDED}) \land (\mathtt{Waypoint} = \emptyset) \rightarrow (\mathtt{Pos}_t = \mathtt{Pos}_{t-1}) \land (\mathtt{Yaw}_t = \mathtt{Yaw}_{t-1}) \land (\mathtt{ALT}_t = \mathtt{ALT}_{t-1})\}$
A.SPORT$_1$	In `SPORT` mode, the vehicle must climb as indicated by the `PILOT_SPEED_UP` parameter.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{SPORT}) \rightarrow (\mathtt{Speed\_vertical}_t = \mathtt{PILOT\_SPEED\_UP})\}$
A.RC.FS$_1$	If and only if the vehicle is armed in `ACRO` mode and the throttle input is less than the minimum (`FS_THR_VALUE` parameter), the vehicle must immediately disarm.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{ACRO}) \land (\mathtt{Throttle}_t < \mathtt{FS\_THR\_VALUE}) \rightarrow (\mathtt{Disarm} = \mathtt{on})\}$
A.RC.FS$_2$	If the throttle input is less than `FS_THR_VALUE` parameter, it must change the current mode to the RC fail-safe mode.	T$_3$	$\square\{(\mathtt{Throttle}_t < \mathtt{FS\_THR\_VALUE}) \rightarrow (\mathtt{RC}_{\mathtt{fail}} = \mathtt{on})\}$
A.CHUTE$_1$	Deploying a parachute requires following conditions: (1) the motors must be armed, (2) the vehicle must not be in the `FLIP` or `ACRO` flight modes, (3) the barometer must show that the vehicle is not climbing, and (4) the vehicle's current altitude must be above the `CHUTE_ALT_MIN` parameter value.	T$_2$	$\square\{(\mathtt{Parachute} = \mathtt{on}) \rightarrow (\mathtt{Armed} = \mathtt{true}) \land (\mathtt{Mode}_t \neq \mathtt{FLIP/ACRO}) \land (\mathtt{ALT}_t \leq \mathtt{ALT}_{t-1}) \land (\mathtt{ALT}_t > \mathtt{CHUTE\_ALT\_MIN})\}$
A.GPS.FS$_1$	When the number of detected GPS satellites is less than four, the vehicle must trigger the GPS fail-safe mode.	T$_3$	$\square\{(\mathtt{GPS}_{\mathtt{fail}} = \mathtt{on}) \rightarrow (\mathtt{GPS}_{\mathtt{count}} < 4)\}$
A.GPS.FS$_2$	When the GPS fail-safe mode is triggered and there is a secondary altitude sensor, the vehicle must change the current primary altitude source to the secondary sensor.	T$_3$	$\square\{(\mathtt{GPS}_{\mathtt{fail}} = \mathtt{on}) \land (\mathit{Baro} = \mathit{on}) \rightarrow (\mathtt{ALT}_{\mathtt{src}} = \mathtt{Baro})\}$
PX.RTL$_1$	If the current altitude is less than `RTL_RETURN_ALT`, then the altitude must be increased until the altitude is greater or equal to the `RTL_RETURN_ALT`.	T$_3$	$\square\{(\mathtt{ALT}_t < \mathtt{RTL\_RETURN\_ALT}) \land (\mathtt{Mode}_t = \mathtt{RTL}) \rightarrow (\mathtt{ALT}_{t-1} < \mathtt{ALT}_t)\}$
PX.RTL$_2$	If the current altitude is greater or equal to `RTL_RETURN_ALT`, current flight mode is `RTL`, and the current vehicle is not at `home position`, then the vehicle must move to the `home position` while maintaining the current altitude.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{RTL}) \land (\mathtt{ALT}_t \geq \mathtt{RTL\_RETURN\_ALT}) \land (\mathtt{Pos}_t \neq \mathit{home\,position}) \rightarrow (\mathtt{Pos}_{t-1} \neq \mathtt{Pos}_t) \land (\mathtt{ALT}_{t-1} = \mathtt{ALT}_t)\}$
PX.RTL$_3$	If current altitude is greater or equal to `RTL_RETURN_ALT` and current position is the same as `home position`, then flight mode must be `LAND`.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{RTL}) \land (\mathtt{ALT}_t \geq \mathtt{RTL\_RETURN\_ALT}) \land (\mathtt{Pos}_t = \mathit{home\,position}) \rightarrow (\mathtt{Mode}_t = \mathtt{LAND})\}$
PX.RTL$_4$	If `RTL_LAND_DELAY` parameter has -1, the vehicle must hover at `RTL_DESCEND_ALT`.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{RTL}) \land (\mathtt{RTL\_DESCEND\_ALT} = -1) \rightarrow (\mathtt{Pos}_t = \mathtt{Pos}_{t-1}) \land (\mathtt{ALT}_t = \mathtt{ALT}_{t-1})\}$
PX.RTL$_5$	It is the same as A.RTL$_4$	T$_3$	It is the same as A.RTL$_4$.
PX.ORBIT$_{1-4}$	It is the same as A.CIRCLE$_{1-4}$.	T$_3$	It is the same as A.CIRCLE$_{1-4}$.
PX.ORBIT$_5$	The maximum radius must be 100 meters.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{ORBIT}) \rightarrow (\mathtt{Circle\_radius}_t < 100)\}$
PX.ORBIT$_6$	The maximum acceleration must be limited to $2m/s^2$.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{ORBIT}) \rightarrow (\mathtt{Circle\_speed}_t < 2m/s^2)\}$
PX.LAND$_1$	Descending speed must be the same as `MPC_LAND_SPEED` parameter.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{LAND}) \rightarrow (\mathtt{Speed\_vertical}_t = \mathtt{MPC\_LAND\_SPEED})\}$
PX.ALTITUDE$_1$	It is the same as A.ALT_HOLD$_2$.	T$_3$	It is the same as A.ALT_HOLD$_2$.
PX.POSITION$_1$	The vehicle must maintain a constant position.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{POSITION}) \rightarrow (\mathtt{Pos}_t = \mathtt{Pos}_{t-1})\}$
PX.HOLD$_1$	It is the same as A.LOITER$_1$.	T$_3$	It is the same as A.LOITER$_1$.
PX.HOLD$_2$	If `MIS_LTRMIN_ALT` is not -1 and current altitude is less than the parameter value, then the vehicle must ascend to this altitude.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{HOLD}) \land (\mathtt{MIS\_LTRMIN\_ALT} \neq -1) \rightarrow (\mathtt{ALT}_t > \mathtt{ALT}_{t-1})\}$
PX.TAKEOFF$_1$	When the vehicle conducts a taking off command, the target altitude must be the `MIS_TAKEOFF_ALT` parameter value.	T$_3$	$\square\{(\mathtt{Command}_t = \mathtt{takeoff}) \rightarrow (\mathtt{ALT}_t \leq \mathtt{MIS\_TAKEOFF\_ALT})\}$
PX.TAKEOFF$_2$	When the vehicle conducts a taking off command, the speed of ascent must be the `MPC_TKO_SPEED` parameter value.	T$_3$	$\square\{(\mathtt{Command}_t = \mathtt{takeoff}) \rightarrow (\mathtt{Speed\_vertical}_t = \mathtt{MPC\_TKO\_SPEED})\}$
PX.GPS.FS$_1$	If time exceeds `COM_POS_FS_DELAY` seconds after GPS loss is detected, the GPS fail-safe must be triggered.	T$_1$	$\square\{(\mathtt{GPS}_{\mathtt{loss}} = \mathtt{on}) \rightarrow (\Diamond_{[0,\mathtt{COM\_POS\_FS\_DELAY}+k]}\mathtt{GPS}_{\mathtt{fail}} = \mathtt{on})\}$
PX.GPS.FS$_2$	If the GPS fail-safe is triggered and a remote controller is available, the flight mode must be changed to `ALTITUDE` mode.	T$_3$	$\square\{(\mathtt{GPS}_{\mathtt{fail}} = \mathtt{on}) \land (RC_t = on) \rightarrow (\mathtt{Mode}_t = \mathtt{ALTITUDE})\}$
PX.GPS.FS$_3$	If the GPS fail-safe is triggered and a remote controller is not available, the flight mode must be changed to `LAND` mode.	T$_3$	$\square\{(\mathtt{GPS}_{\mathtt{fail}} = \mathtt{on}) \land (RC_t = off) \rightarrow (\mathtt{Mode}_t = \mathtt{LAND})\}$
PP.Hover	The vehicle must be staying in a constant position and heading.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{Hover}) \rightarrow (\mathtt{Pos}_t = \mathtt{Pos}_{t-1}) \land (\mathtt{Yaw}_t = \mathtt{Yaw}_{t-1})\}$
PP.Hover$_Z$	It is the same as A.ALT_HOLD$_2$.	T$_3$	It is the same as A.ALT_HOLD$_2$.
PP.Hover$_C$	It is the same as A.LOITER$_1$.	T$_3$	It is the same as A.LOITER$_1$.
PP.TAKEOFF$_1$	When the vehicle conducts a taking off command, the target altitude must be 5 more meters than home altitude.	T$_3$	$\square\{(\mathtt{Command}_t = \mathtt{takeoff}) \rightarrow (\mathtt{ALT}_t \leq \mathtt{HOME\_ALT} + 5)\}$
PP.HOME$_1$	The vehicle must descend while it moves to home position.	T$_3$	$\square\{(\mathtt{Mode}_t = \mathtt{HOME}) \land (\mathtt{Land}_t \neq \mathtt{true}) \rightarrow (\mathtt{ALT}_t \neq \mathtt{ALT}_{t-1}) \land (\mathtt{Pos}_t \neq \mathtt{Pos}_{t-1})\}$

TABLE XII: Policies extracted from the docs and comments on source code of ArduPilot (A) [10], PX4 (PX) [52], and Paparazzi (PP) [32].