PGPATCH: Policy-Guided Logic Bug Patching for Robotic Vehicles

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

***Abstract*—Automated program repair (APR) methods aim to identify patches for a given bug and apply them with minimal human intervention. To date, existing APR approaches focus on repairing software bugs, such as memory safety bugs. However, our analysis of popular robotic vehicle (RV) control software shows that most of their bugs are not memory bugs but rather logic bugs. These bugs, while not causing software crashes, can cause an RV to reach an undesired physical state (e.g., hitting the ground).**

To fix these logic bugs, we introduce PGPATCH, a policy-guided program repair framework for RV control programs, which identifies the correct patch for a given logic bug and applies it without human intervention. PGPATCH takes, as input, existing or new logic formulas used to discover logic bugs. It then leverages the formulas using a dedicated dynamic analysis to classify the previously known logic bugs into a patch type. It next uses a customized algorithm, based on the identified patch type and violated formula, to produce a source code patch as output. Lastly, it creates repeatable tests to verify the patch's completeness, ensuring that the patch is correct and does not degrade the RV's performance. We evaluate PGPATCH on selected bug cases from three popular RV control software and find that it correctly fixes 258 out of 297 logic bugs (86.9%). We additionally recruit 18 experienced RV developers and users and conduct a user study that demonstrates how using PGPATCH makes fixing bugs in RV software significantly quicker and less error-prone.

I. INTRODUCTION

Patching security vulnerabilities in a timely manner is crucial to prevent attackers from compromising software. Yet, recent works have shown that creating patches on time is challenging due to the required manual effort [33], [75]. For this reason, there has been an increasing interest in automated program repair (APR) to create patches without human intervention.

There are two broad types of APR approaches. First, test suite-based methods generate patches for bugs that cause outputs to deviate from developers' expectations [20], [27], [40], [49], [54], [56], [79], [80]. Their patch localization and generation algorithms require users to input passing and failing test cases that define the program's expected outputs corresponding to certain inputs (i.e., test suites) [39], [46]. Second, specification-based APR approaches fix bugs through specifications that define functional requirements in the form of natural language documents or formal logic formulas [5], [15], [34], [37], [38], [48], [82].

In this paper, we consider logic bugs in robotic vehicle (RV) control software, which are bugs that cause deviations in the RV's physical behavior from the developer's expectations but do not cause the program to stop execution [32]. The reason is that our analysis of 1,257 existing bugs (from 2014 to 2021) in two popular RV control programs (ArduPilot [8] and PX4 [64]) indicates 98.2% of them are logic bugs, showing the prevalence of logic bugs in RV software. Logic bugs in RVs mainly stem from misimplementations and design flaws. For example, PX4's documentation states that "*fail-safe mode* must be triggered when GPS loss is detected" [28]. However, PX4 fails to trigger the *fail-safe mode* under the following three conditions: (1) the `COM_POS_FS_DELAY` configuration parameter has a negative value, (2) the RV is in the `CIRCLE` flight mode, and (3) the RV passes through an area where the GPS signal is not available. This logic bug leads to unsafe states, which causes the RV to float unpredictably, potentially crashing into an obstacle.

Promptly fixing bugs in RVs is critical because an attacker can stealthily exploit such logic bugs to cause undesired behaviors and physical damage [17], [41]–[43]. Yet, patching logic bugs in RVs is more challenging than fixing bugs in traditional software for two main reasons. First, the "correct behavior" of RVs depends not only on the "cyber space" (i.e., how RVs behave according to inputs given to control software) but also on the "physical space" (i.e., the physical environment in which they operate). For example, consider that a user gives an RV input to make the RV move forward. From a traditional software point of view, the RV's correct behavior is to move forward because the software must show consistent behaviors according to program inputs. Yet, the RV may show different correct behaviors based on the current physical environment: (*i*) moving backward if the RV is near an obstacle, (*ii*) landing on the ground if the RV loses GPS signals, and (*iii*) staying in a stable position if a strong wind blows. Second, the "input space" and "output space" of RV software are much larger than those of traditional software, given the amount of data that an RV receives and processes at any given point in time. This data encompasses periodic measurements from multiple sensors as well as commands given by a ground control station (GCS).

Unfortunately, existing APR approaches do not address these challenges. In fact, test suite-based methods fail to fix logic bugs in RV control programs because the correctness of patches depends on the test suite's completeness. Achieving completeness is challenging due to the RV software's large input and output space, as shown by previous work [45], [52], [71]. Further, specification-based approaches cannot fix logic bugs since they mainly focus on memory bugs [34], [48], or they do not create code-level patches from specifications that define the software's correct behavior [5], [15], [37], [38], [82].

To address these limitations, in this paper, we introduce

PGPATCH, a policy-guided program repair framework for RV control programs, which generates patches for a given logic bug and applies them without human intervention. PGPATCH is composed of four interconnected components: (1) Preprocessor, (2) Patch Type Analyzer, (3) Patch Generator, and (4) Patch Verifier. First, the Preprocessor takes an input that triggers the bug and a formula written in the PGPATCH policy language (PPL) which defines RVs' expected operations. This component checks the validity of the formula given by users. We note that PGPATCH requires only a single test case triggering the bug (i.e., one failing test case) while test suite-based APR methods require a complete set of test cases, including both failing and passing cases. Second, to handle diverse types of logic bugs in RVs, the Patch Type Analyzer finds the most appropriate patch type to fix the specific logic bug. Third, the Patch Generator component finds the proper patch location for the targeted bug and creates a patch. PGPATCH first identifies a patch location through the PPL formula and pattern matching. It then creates a patch based on the formula, identified patch type, and patch location. Lastly, the Patch Verifier inserts the created patch into the target source code and compiles the patched code. It then tests if the patch fixes the bug and confirms that the patch does not break the RV's functionality or degrade its performance.

We evaluate PGPATCH on ArduPilot [8], PX4 [64], and Paparazzi [59], the three most popular flight control software used in many commodity RVs. PGPATCH correctly fixes 258 out of 297 logic bugs (86.9%) requiring, on average, 12.5 minutes to create one patch. Further, we conduct a user study to compare the effort required by RV software developers and users to build PPL formulas with their effort to create manual code-level patches. Our user study shows that building PPL formulas is easier and less error-prone compared to manually patching logic bugs. We make the following contributions:

- **Behavior-aware Patch Generation.** We introduce PGPATCH, a policy-guided APR framework for RV control programs, which leverages existing or new logic formulas for patch localization and generation via a combination of static and dynamic analysis. PGPATCH also creates repeatable tests to validate the patch's correctness.
- **Evaluation with Real-world RV Software.** We applied PGPATCH to the three most popular RV control software packages. PGPATCH generated correct patches for 258 of the 297 previously known bug cases (86.9%).
- **User Study.** We recruited experienced RV developers and users and conducted a user study that demonstrates the usefulness of PGPATCH for patching bugs compared to manual patching.

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

Ethical Considerations and Responsible Disclosure. We responsibly disclosed any previously unknown bug discovered in this paper to the affected RV software developers. In our user study, we avoid collecting any personally identifiable information (PII). Our study was reviewed by our institution's IRB and considered IRB exempt.

```
<formula>      ::= <term> <verb> <value> | <conjunction>
                   <formula> | if <formula>, then <formula>
                   | iff <formula>, then <formula>
<term>         ::= S | P | E | V | F
<verb>         ::= `is' | `is not' | `is more than' |
                   `is less than' |
                   `is greater than or equal to' |
                   `is less than or equal to'
<value>        ::= S | P | V | F |
                   `true' | `false' | `enabled' |
                   `disabled' | `error' | `on' |
                   `off' | <integer> | <real_number>
<conjunction> ::= `and' | `or'
```

Listing 1: PGPATCH policy language (PPL) syntax in BNF.

II. PRELIMINARIES

Logic Bugs and Adversarial Exploitation in RVs. In this paper, we use the term "logic bugs" to refer to bugs that cause a program to operate incorrectly, leading to undesired physical behavior, without causing a program crash or memory corruption. Starting from this definition, we consider buffer overflows, null pointer dereferences, and divisions by zero as non-logic bugs [41], [43]. Logic bugs are caused by developers incorrectly designing or implementing software components. The developers' mistakes might occur for various reasons, including but not limited to unexpected environmental conditions (e.g., strong wind and significant sensor noise) and copying-and-pasting a buggy code snippet [16], [19], [72].

We assume that an adversary can exploit a logic bug to stealthily disrupt an RV's normal behaviors. Particularly, the RV's three types of inputs (configuration parameters, user commands, and environmental factors) can be leveraged to trigger a logic bug. The adversary can (1) override the configuration parameter values, (2) replay or spoof user commands, and (3) change environmental conditions or wait for suitable conditions before conducting attacks. The adversary can conduct input manipulation by exploiting known vulnerabilities in the RV's communication protocol and sensors [44], [73], [81].

PGPATCH Policy Language (PPL). A recent work on logic bug-finding (PGFuzz [41]) has created linear temporal logic (LTL) formulas for discovering logic bugs. Particularly, users identify the RVs' correct behaviors through official documentations [11], [59], [68] and formally represent them with LTL templates [41], [82]. However, defining LTL formulas requires users to learn about syntax and rules of temporal logic. To reduce the difficulty of LTL formulas, we introduce PPL in BNF notation, as shown in Listing 1. In PPL formulas, the "term" can be formed over an RV's physical state (`S`), configuration parameter (`P`), environmental factor (`E`), the name of a variable (`V`), and the name of a function (`F`).

Additionally, we design two PPL templates: (*i*) T_1: "[term] [verb] [value] (conjunction)" and (*ii*) T_2: "if / iff [term] [verb] [value] (conjunction), then [term] [verb] [value] (conjunction)". Users can use these formula templates to easily and quickly build new formulas. For example, given a policy in natural language stating that "the engine must be turned on", users can express the policy with T_1 template: "engine is on". Here, "engine", "is", and "on" are matched with "term", "verb", and "value" keywords. These three types of keywords compose

```
1  // Get a time delay to trigger position fail-safe
2  param_get(param_find("COM_POS_FS_DELAY"), &val);
3  // Force the valid range of the parameter
4  posctl_nav_loss_delay = math::constrain(val * sec_to_usec,
5  POSVEL_PROBATION_MIN, POSVEL_PROBATION_MAX);
```

Listing 2: GPS Fail-Safe Bug [41].

```
1 bool AP_Arming_Rover::pre_arm_checks() {
2   if (rover.g2.sailboat.sail_enabled()
3     && !rover.g2.windvane.enabled()) {
4       printf("Sailing enabled with no WindVane");
5       return false;
```

Listing 3: Sailboat Pre-Arming Bug [1].

a proposition that is mandatory to build a PPL formula. Conversely, the "conjunction" is optional. T_2 template is for expressing RV behaviors that have preconditions and post-conditions in PPL formulas. In T_2 template, "if" expresses imply and "iff" represents "if and only if", where "if" is used for unidirectional statements and "iff" is used for bidirectional statements. For instance, given a policy stating that "if the RV lands on the ground and the pilot turns on disarming, then the engine must be turned off", the users can represent this policy with T_2 template: "if land is true and disarm is on, then engine is off". Here, "land is true" and "disarm is on" are preconditions to trigger a post-condition ("engine is off"). Lastly, we implement a translator that converts LTL to PPL formulas so that PGPATCH can leverage the existing LTL formulas from logic bug-finding tools to fix bugs (Section V-B).

III. Examples of Logic Bugs

A. GPS Fail-Safe Bug

PX4 documentation states that "If the time exceeds `COM_POS_FS_DELAY` seconds after GPS loss is detected, the GPS fail-safe must be triggered". This policy is formally expressed in PPL syntax as: "If $\texttt{GPS}_{\texttt{loss}}$ is `on` and $\texttt{Loss}_{\texttt{time}}$ is more than `COM_POS_FS_DELAY`, then $\texttt{GPS}_{\texttt{fail}}$ is `on`". PX4 v1.7.4 forces the `COM_POS_FS_DELAY` configuration parameter to have a value in the valid range (from 1 to 100) at lines 4 and 5 in Listing 2. However, developers remove these lines from PX4 v1.9. If users assign a value outside the valid range (e.g., -1) to the parameter, it causes PX4 to fail to trigger the GPS fail-safe and randomly fly in the air when it loses GPS signals [41].

B. Sailboat Pre-Arming Bug

ArduPilot documentation states that "pre-arming must return an error when a sailboat is turned on without a wind vane", formally expressed with the formula: "If `armed` is `false` and `SAIL_ENABLE` is `1` and `WNDVN_TYPE` is `0`, then `pre_arm_checks` is `error`". This policy intends that the RV software must not allow the sailboat to operate without the wind vane because RVs cannot navigate a waypoint without wind direction obtained from the wind vane. However, ArduPilot did not implement this, causing the RV to deviate from its planned path [1]. To fix this bug, developers add an "if statement" into the source code, as shown in Listing 3.

```
1 void Copter::failsafe_battery_event(void) {
2   if (ap.land_complete)
3     // Stop motors
4   else if (g.failsafe_battery_enabled == FS_BATT_RTL
5     && home_distance > wp_nav.get_wp_radius())
6     // Switch to RTL
7   else // Switch to LAND
```

Listing 4: Battery Fail-Safe Bug [13].

```
1 void FlightTaskAutoMapper::_prepareLandSetpoints() {
2   _constraints.tilt = _param_mpc_tiltmax_lnd.get();
3 ...
4 bool FlightTaskManualAltitude::activate() {
5   _constraints.tilt = _param_mpc_man_tilt_max.get();
```

Listing 5: Tilt Angle Bug [77].

C. Battery Fail-Safe Bug

When a fail-safe condition is detected (e.g., due to a low-battery condition), and the RV is within 2 meters from its home location, ArduPilot must change the flight mode to the `LAND` mode, making the RV decrease its altitude and land on the ground. If the RV is farther than 2 meters from its home location, ArduPilot must switch the flight mode to the `RTL` mode. This mode first makes the RV increase its altitude by `RTL_ALT` parameter value, then, it makes the RV navigate to its home position and land on the ground. This is represented in PPL syntax as: "If `Failsafe` is `on` and `FS_BATT_ENABLE` is `2` and `home_distance` is more than `2`, then `mode` is `RTL`". However, ArduPilot decides its flight mode based on the `WPNAV_RADIUS` parameter, instead of the hard-coded 2 meters as depicted at line 5 in Listing 4. This causes the RV to operate in an incorrect flight mode.

D. Tilt Angle Bug

RV control programs typically limit a tilt range to prevent drastic behaviors. However, the programs must not limit tilt in `LAND` or `RTL` flight modes, as this may cause the RV to lose position control when it is descending. A policy for this behavior is defined as: "If `mode` is `LAND` or `mode` is `RTL`, then `_constraints.tilt` is `disabled`". However, PX4 limits the tilt value in `LAND` and `RTL` modes [77], which violates the above PPL formula, as shown in Listing 5.

E. Exploiting Logic Bugs

Attackers can exploit logic bugs to perform hard-to-detect attacks. For instance, an attacker can exploit the Battery Fail-Safe Bug (Section III-C) to crash an RV on the ground, by assigning 1 to `WPNAV_RADIUS` and 300 to `RTL_ALT`, the `RTL` mode's minimum altitude. Then, if the RV's location is 2 meters from its home location, and the fail-safe is triggered, ArduPilot decides to change the current flight mode to `RTL` mode, since the distance between the RV and the home location is greater than the `WPNAV_RADIUS` parameter value (i.e., 1 meter). Then, the RV keeps increasing its altitude by 300 meters. This altitude increase can completely deplete the RV's battery (especially if the fail-safe was triggered due to a low-battery condition), leading the RV to a physical crash. Similarly, the adversary can exploit the other logic bugs introduced in Section III.

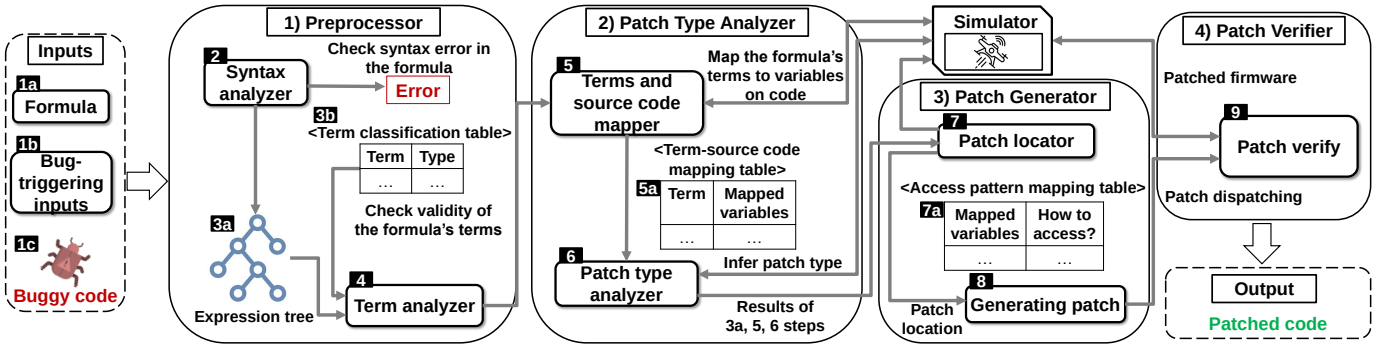


Fig. 1: Overview of PGPATCH's workflow and architecture.

IV. Logic Bug Analysis

As a preliminary analysis, by analyzing 1,257 existing bugs in ArduPilot and PX4 from 2014 to 2021 (Detailed in Appendix A), we found that 33.7% of them can be fixed with one of the following five patch categories: (1) disabling a statement, (2) checking valid ranges of parameters, (3) updating a statement, (4) adding a condition check, and (5) reusing an existing code snippet. Therefore, we designed PGPATCH to address these patch types. The other 66.3% of the patches require (1) adding third party libraries, (2) implementing a new feature from scratch, (3) defining a new variable, loop, or function, and (4) other complicated techniques, e.g., adding/updating mathematical formulas. These requirements prevent PGPATCH from automatically creating patches for these bugs (Detailed in Section VII-B).

Disabling a Statement (`DISABLE`). A logic bug may cause an RV to change its physical state when it is not supposed to. This mainly occurs when developers miss prohibiting a behavior. Fixing this bug type requires localizing and disabling the statement that changes the RV's state incorrectly.

Checking Valid Ranges of Parameters (`CHECK`). RV software may fail to check the valid ranges of some configuration parameters, which leads to unexpected behaviors (e.g., instability or loss of attitude control). When a parameter has a value outside the valid range, its value must be restored to its default setting to fix such bugs.

Updating a Statement (`UPDATE`). When an RV's states do not satisfy all of the preconditions to trigger a behavior, the RV software must not trigger the behavior. However, an incorrect "if statement" may allow the RV software to trigger the behavior even though the RV's states satisfy only a part of the preconditions. To fix this type of bug, the "if statement" must be updated with the correct preconditions.

Adding a Condition Check (`ADD`). A missed condition check (i.e., a missing "if statement") might prevent RVs from conducting correct behaviors even though all the preconditions are satisfied. Hence, adding the correct condition check and triggering the correct behavior fix such bugs.

Reusing an Existing Code Snippet (`REUSE`). Some logic bugs may cause the RV software to stop checking an RV's states after a specific flight stage (e.g., takeoff). These can be patched by reusing the existing code snippets in all the flight stages.

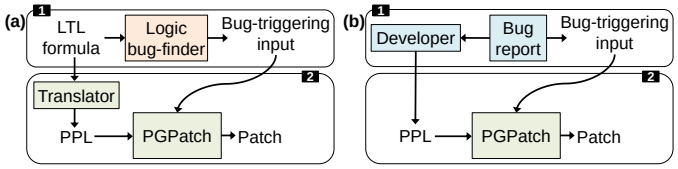


Fig. 2: Two usage scenarios of PGPATCH.

V. PGPatch

A. System Overview

PGPATCH is a policy-based program repair tool for RVs, which patches logic bugs in the source code. Figure 1 demonstrates its four interconnected components, and Figure 2 shows its two different usage scenarios. Particularly, PGPATCH takes, as input, (1) a formula, which defines the RV's expected operation, and (2) a bug-triggering input that includes user commands, configuration parameters, environmental factors, and the RV's physical states (e.g., the battery level). The formula can be given to PGPATCH as input in two different ways. (*i*) Figure 2-(a) shows the main usage scenario of PGPATCH. It takes existing LTL formulas, which were used to discover logic bugs through a logic bug-finding tool (e.g., PGFuzz [41]). PGPATCH converts the LTL formulas to PPL formulas, and it uses PPL formulas to create patches. We note that LTL syntax includes several temporal relation operators [41], [82]. Yet, most requirements in RV documentation can be described by "always" operator (e.g., all formulas in PGFuzz are in the form of "always"). Based on this observation, we designed PPL formulas (Section II) to only support "always" operator. (*ii*) Figure 2-(b) shows how PGPATCH allows developers to define PPL formulas to fix logic bugs obtained from bug reports. In this scenario, the bugs are not discovered using LTL formulas.

Preprocessor. This component translates LTL formulas to PPL formulas if users input LTL formulas. It then verifies the syntax and semantics of the formulas and classifies the formulas' terms to a variable, function, or an RV's state types.

PGPATCH (❷) leverages PPL formulas, which define an RV's correct behavior (❶a). For instance, we represent the correct sailboat pre-arming behavior (Section III-B) with: "If `armed` is `false` and `SAIL_ENABLE` is `1` and `WNDVN_TYPE` is `0`, then `pre_arm_checks` is `error`". PGPATCH starts its analysis by creating an expression tree of the PPL formula (❸a). It next collects the formula's terms by visiting the terminal nodes of the expression tree. It then uses a term classification table (❸b)

to classify each term into a code variable, function, or the RV's state types (❹). For instance, in the sailboat's formula, `armed` is an RV's physical state. `SAIL_ENABLE` and `WNDVN_TYPE` are the RV's configuration parameter states, and the `pre_arm_checks` is a function in the source code.

Patch Type Analyzer. This component first maps the formula's terms to the variables and functions in the source code through static analysis. It then verifies the mapping's correctness through dynamic analysis (❺) and creates a term-source code mapping table (❺a). For example, PGPATCH maps the `WNDVN_TYPE` to the integer type `direction` variable of the `AP_WindVane` class. It then changes the `WNDVN_TYPE` parameter on an RV simulator and checks whether the `direction` variable has the same value as the changed `WNDVN_TYPE` parameter value to confirm the mapping is correct. PGPATCH then finds the required patch type to fix a specific logic bug since each patch type requires a different method for patch location identification and patch creation.

PGPATCH classifies each logic bug into one of the five patch types: (1) `ADD`, (2) `REUSE`, (3) `UPDATE`, (4) `DISABLE`, and (5) `CHECK` (❻). For this, it leverages (*i*) the preconditions that must be satisfied to trigger the bug, and (*ii*) whether the bug occurs only at a specific flight stage (e.g., after takeoff) or at all flight stages. For instance, PGPATCH classifies Sailboat Pre-Arming Bug (Section III-B) to `ADD` patch type because: (1) the RV control software does not raise an error even though the RV's states satisfy the preconditions to produce the desired error message, and (2) the error message is not returned regardless of flight stages (Detailed in Section V-C). This hints that the developers miss inserting an "if statement" to raise the error message, resulting in the bug.

Patch Generator. This component finds the patch location and generates the patch. PGPATCH identifies the patch location using a pattern matching approach (❼), which matches the code locations to the formula's terms. For example, in the sailboat policy, the left side of "then" is the precondition to trigger a post-condition, raising a `pre_arm_checks` error. PGPATCH obtains the `bool pre_arm_checks()` function as the potential patch location because the post-condition (`pre_arm_checks`) is mapped to it. It next creates an access pattern mapping table representing how to access the variables and functions from the inferred patch location (❼a). For example, the code snippet in the `AP_Arming` class calls the `rover.g2.windvane.enabled()` function to access the `direction` variable in the `AP_WindVane` class. It then creates a patch using the identified patch type, access pattern mapping table, and location (❽). For example, it produces the following code to fix the sailboat's *pre-arming* misbehavior: "*if (rover.g2.sailboat.sail_enable() == true && rover.g2.windvane.enabled() == false){return false;}*".

Patch Verifier. The last component is the Patch Verifier, which deploys the created patch into the target source code and creates a binary executable file by compiling the patched source code. It then tests the patch using the simulator to see whether the patch fixes the bug and does not interfere with the RV's intended functionality and performance (❾).

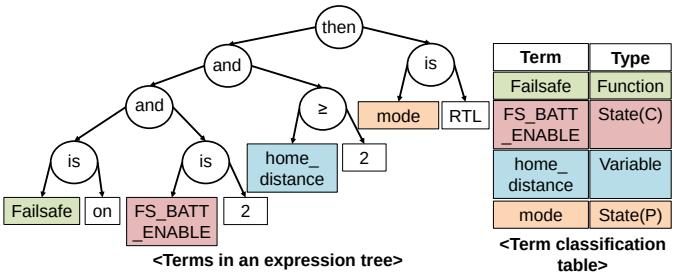


Fig. 3: Illustration of an expression tree constructed from Battery Fail-safe formula (Section III-C) and term classification table. *State (P)* and *State (C)* denote a physical state and configuration parameter state, respectively.

B. Preprocessor

The Preprocessor translates LTL formulas to PPL formulas if users input LTL formulas to PGPATCH. It then verifies PPL formulas' syntax and classifies the formulas' terms to variables, functions, or RV's states. To do so, it uses a term classification table (❸b Figure 1). The table's each row consists of a term and type, where the type can be a variable, function, or state, as shown in Figure 3.

1) Checking Syntax Errors in PPL Formulas: PGPATCH creates an expression tree of a given PPL formula and checks whether syntax errors exist in the formula. If it detects any syntax error, it refuses to generate a patch. If there are no errors, it converts the formula to an expression tree and continues with the next steps. To illustrate, the expression tree of the Battery Fail-safe's formula[1] is presented in Figure 3.

2) Checking Semantic Errors in PPL Formulas: Users could unintentionally input formulas that contain semantic errors, e.g., a formula that contradicts another existing formula. To resolve this problem, the Preprocessor verifies whether an added formula contradicts any existing formulas. For instance, if a user inputs the formula: "if $\texttt{State}_i$ is `true`, then $\texttt{Behavior}_i$ is `on`", PGPATCH can detect that this formula contradicts the following existing formula: "if $\texttt{State}_i$ is `true`, then $\texttt{Behavior}_i$ is `off`". In this case, PGPATCH produces an error message and denies creating a patch corresponding to the newly added formula.

3) Checking Validity of Terms: In this step, PGPATCH achieves two goals: (*i*) classifying terms into variable, function, or RV state types, and (*ii*) detecting any syntax error in the terms. PGPATCH first classifies each term into "variable", "function", or "RV state" types based on the term classification table, as shown in Figure 1 (❸b) and Figure 3. PGPATCH creates the table from RV source code and documentation (Detailed in Section VI). If a term does not match any type in the term classification table, PGPATCH considers it as a syntax error and stops generating the patch.

C. Patch Type Analyzer

PGPATCH maps terms to variables/functions in the source code to determine the most appropriate patch type to fix a bug.

[1]If `Failsafe` is `on` and `FS_BATT_ENABLE` is `2` and `home_distance` is more than `2`, then `mode` is `RTL`

Term	Mapped variables / functions on source code
failsafe	~~void Copter::***failsafe_ekf_event()*** in ekf_check.cpp~~ ~~void Copter::***failsafe_ekf_off_event()*** in ekf_check.cpp~~ void Copter::***failsafe_battery_event()*** in events.cpp void Copter::***failsafe_gcs_check()*** in events.cpp …
FS_BATT_ENABLE	int ***failsafe_battery_enabled*** in Parameters.cpp
home_distance	~~float ***home_distance*** in AC_Fence.h~~ int ***home_distance*** in Copter.h

Fig. 4: Illustration of a term-source code mapping table.

1) Mapping Terms to Source Code: PGPATCH infers each term's type as detailed in Section V-B3. However, finding a patch location and generating the patch requires mapping the terms to specific variables/functions in the source code. To address this, PGPATCH matches each term with variables/functions in the source code via name-based matching. Specifically, when the term and the variable have the same name, they are matched. For instance, PGPATCH extracts *Failsafe*, *FS_BATT_ENABLE*, and *home_distance* terms from the created expression tree of Battery Fail-safe (Figure 3). It then constructs a term-source code mapping table (Figure 4), which consists of each term and mapped variables/functions.

We found that a term can be mapped to multiple variables that have the same name in the source code. For example, the *home_distance* term is mapped to two different variables, `float home_distance` and `int home_distance`, as they have the same variable name. Yet, not all mapped variables are related to the logic bugs, e.g., `float home_distance` is not related to the *home_distance* term to trigger the bug. To filter the unrelated variables, PGPATCH performs a dynamic analysis on an RV simulator: (1) it annotates the mapped variables in the source code, (2) it compiles the annotated code and uploads a new binary file into the simulator, (3) it executes the inputs that trigger a logic bug, e.g., moving an RV to another location, and (4) it filters out the variables that do not change their value based on the executed inputs. For example, the `float home_distance` variable always has a zero value regardless of the executed bug-triggering inputs because ArduPilot changes this value only if geo-fence is activated [25]. Thus, PGPATCH excludes the `float home_distance` variable from the term-source code mapping table (Figure 4).

2) Identifying a Patch Type: Different logic bugs require a separate technique for inferring their patch locations and generating the patches. To identify the patch type that fixes the logic bug, PGPATCH first executes inputs to trigger the logic bug on the simulator and collects the following run-time information: (1) the violated PPL formula's propositions for `DISABLE`, (2) the configuration parameters' values for `CHECK`, (3) the preconditions that trigger a logic bug for `UPDATE`, and (4) the flight stage at which the bug occurs for `ADD` and `REUSE`. PGPATCH then extracts preconditions and post-conditions from the violated formula. For instance, in the PPL formula "if $\texttt{State}_\texttt{i}$ is `true`, then $\texttt{Behavior}_\texttt{i}$ is `on`", the precondition is the left side of the "then" keyword, i.e., ($\texttt{State}_\texttt{i} = \texttt{true}$), and the post-condition is the right side of "then", i.e., ($\texttt{Behavior}_\texttt{i} = \texttt{on}$). Using this information, PGPATCH determines the patch type by traversing the flow diagram (Figure 12 in Appendix).

	Flight stages
Test Case 1	1) ANGLE_MAX = 5 2) Arming 3) Takeoff 4) Navigating waypoints
Test Case 2	1) Arming 2) ANGLE_MAX = 5 3) Takeoff 4) Navigating waypoints

Fig. 5: An example of test cases created from Algorithm 1.

Disabling a Statement (`DISABLE`). PGPATCH classifies the logic bug to be patched as `DISABLE` type when a violated PPL formula and the RV's states satisfy the following two conditions: (*i*) the formula's post-condition explicitly represents a state that must not be changed, and (*ii*) the RV's states satisfy the formula's preconditions but violate the post-conditions. This is because disabling a statement can prevent triggering the incorrect state change that violates the formula's post-condition. For instance, for the Tilt Angle Bug (Section III-D), PX4 limits the tilt value during the `LAND` and `RTL` flight modes, violating the formula[2]. PGPATCH classifies this bug into `DISABLE` because the logic bug causes a change in *_constraints.tilt* during `LAND` or `RTL` modes even though it must not be changed according to the policy's post-condition (i.e., `_constraints.tilt` = `disabled`).

Checking Valid Parameter Ranges (`CHECK`). PGPATCH classifies logic bugs as `CHECK` when an out-of-range parameter value causes a policy violation. PGPATCH obtains the valid values of parameters from the RV's documentation and checks whether an input, which contains a value outside its valid range, causes a policy violation. For example, `COM_POS_FS_DELAY` parameter's valid range is from 1 to 100. Yet, PX4 does not check the parameter's value, which causes PX4 to fail to trigger the GPS fail-safe when it loses GPS signals. PGPATCH classifies this bug as `CHECK` because the parameter contains a value outside its valid range.

Updating a Statement (`UPDATE`). PGPATCH classifies a logic bug into the `UPDATE` patch type if an RV's states do not satisfy all of the preconditions in a PPL formula before triggering a behavior. The reason is that an incorrect "if statement" makes the RV prematurely trigger a behavior, which can be fixed by updating the "if statement". For example, ArduPilot conducts pre-flight checks before arming the motors. The pre-flight ensures an RV's current states are ready to start a flight. Yet, the RV software must temporarily stop the pre-flight check when the RV conducts sensor calibration as the pre-flight detects strange sensor values during the calibration and incorrectly concludes that the RV cannot make a fly. This bug violates the following policy: "Iff `armed` is `false` and `calibration` is `false`, then `preflightCheck` is `on`". PGPATCH classifies this case as `UPDATE` because the RV triggers the pre-flight behavior although it satisfies only the "`armed` is `false`" precondition.

Distinguishing between `ADD` and `REUSE`. If a logic bug is not classified as `DISABLE`, `CHECK`, or `UPDATE`, PGPATCH uses a custom algorithm (Algorithm 1 in Appendix) to further determine the patch types of `ADD` and `REUSE`. The algorithm takes two types of inputs: (1) a default mission plan that consists of *arming*, *takeoff*, *navigating waypoints*, *return to home position*, and *landing*. ($\texttt{Mission}_\texttt{set}$) and (2) bug-triggering

[2]If `mode` is `LAND` or `mode` is `RTL`, then `_constraints.tilt` is `disabled`

inputs ($\texttt{Input}_{\texttt{bug}}$). We adopted a default mission plan from ArduPilot to create the $\texttt{Mission}_{\texttt{set}}$[3].

The algorithm works as follows: (1) It creates test cases consisting of user inputs to trigger the bug and the default mission plan. Each test case triggers the logic bug at a different flight stage. (2) It executes a user input of a created test case on a simulator. (3) It checks whether a policy violation (i.e., logic bug) occurs after executing each user input. (4) It logs the RV's physical states when the logic bug occurs. (5) It repeats (2)-(4) for each test case. For instance, an input to trigger the logic bug ($\texttt{Input}_{\texttt{bug}}$) is `ANGLE_MAX = 5`. In the first test case, PGPATCH triggers the logic bug at the first flight stage, as shown in Figure 5. In the second test case, the logic bug is triggered at the second flight stage. PGPATCH triggers the logic bug at all flight stages to determine the patch type.

Adding a Condition Check (`ADD`). If the algorithm outputs that a policy violation occurs regardless of the flight stages, PGPATCH classifies it into `ADD` patch type. This is because an RV's control program does not trigger a behavior though the RV's states satisfy the preconditions to trigger the behavior. For the Sailboat Pre-Arming Bug (Section III-B): "If `armed` is `false` and `SAIL_ENABLE` is `1` and `WNDVN_TYPE` is `0`, then `pre_arm_checks` is `error`", ArduPilot does not produce a pre-arming error when the formula's preconditions are satisfied (i.e., *armed is false*, *SAIL_ENABLE is 1*, and *WNDVN_TYPE is 0*). This means ArduPilot does not have an "if statement" to trigger the behavior; thus, it is classified as `ADD`.

Reusing an Existing Code Snippet (`REUSE`). If the algorithm outputs that an RV does not trigger a behavior after a specific flight stage, we consider such a case as `REUSE`. For instance, ArduPilot must return an error message to the ground control system when `ANGLE_MAX` parameter has a value outside its valid range. This policy is formally expressed as "If `ANGLE_MAX` is less than `1000` or `ANGLE_MAX` is more than `8000`, then `prearm` is `error`". The algorithm's first test case (See Figure 5) does not lead to a logic bug. However, the second test case causes a formula violation because ArduPilot does not check the valid values of `ANGLE_MAX` parameter after the arming flight stage. This bug leads to unstable attitude control or even crashing on the ground. PGPATCH classifies this case into `REUSE` because the logic bug occurs only after the arming flight stage.

D. Patch Generator

To generate a patch, PGPATCH first creates an access pattern mapping table representing how PGPATCH accesses the mapped variables and functions in the term-source code mapping table, and then infers the patch location. PGPATCH uses this table to create and insert the patch.

Below, we first describe how to create the access pattern mapping table (Section V-D1). We then explain how PGPATCH finds the patch location and generates a patch per each patch type (Section V-D2 – Section V-D6).

1) Creating an Access Pattern Mapping Table: To infer how to access the mapped variables and functions in a patch location, PGPATCH conducts the following four steps. First, it merges all source files of an RV control software. Second, it extracts mapped variables and functions from the term-source code mapping table (Figure 4). Third, if a patch location is in the same class as the mapped variables/functions, it directly accesses them. Otherwise, PGPATCH leverages encapsulation in object-oriented programming and uses public getter/setter functions to access the private variables. Specifically, PGPATCH finds *get*, *set*, *enabled*, and *disabled* functions that contain the mapped private variables. To illustrate, when the *WNDVN_TYPE* term is mapped to the `direction` variable (Figure 6), it searches for a function that uses one of the above function names (i.e., either getter or setter) and returns the mapped private variable. It finds the following function that returns the mapped `direction`'s value.

Term	Mapped variables / functions on source code	How to access?
armed	bool *armed* in AP_Notify.h	AP_Notify::flags.armed
SAIL_ENABLE	int *enable* in Sailboat.h	rover.g2.sailboat.sail_enabled()
WNDVN_TYPE	int *direction* in AP_WindVane.h	rover.g2.windvane_enabled()

Fig. 6: An access pattern mapping table for generating a patch.

```
bool AP_WindVane::enabled(){return direction!=WINDVANE_NONE;}
```

Lastly, PGPATCH learns how to access the mapped variables and functions from the merged source code. PGPATCH searches how other classes call the found function, e.g., `enabled()`. For example, other classes in ArduPilot call `enabled()` via `rover.g2.windvane.enabled()`. However, there may be multiple patterns to access a single variable or function from the source code. To choose the correct one, PGPATCH first inserts a found access pattern to a patch location, then, PGPATCH verifies which access pattern is the correct one to access the mapped variable or function.

2) Adding a Condition Check (`ADD`): To generate patches for `ADD` patch type, PGPATCH inserts a missing "if statement" at the right location in the source code.

To find the patch location, PGPATCH extracts terms from a PPL formula's post-condition part. It then maps the terms to variables and functions in the source code, and uses the mapped variables and functions as potential patch location candidates. For example, in the sailboat policy (Section III-B): "If `armed` is `false` and `SAIL_ENABLE` is `1` and `WNDVN_TYPE` is `0`, then `pre_arm_checks` is `error`", the left-hand side of the "then" keyword is the preconditions to trigger an action (i.e., *pre_arm_checks* must return *false*). Further, the *pre_arm_checks* term is mapped to the `bool pre_arm_checks()` function. Thus, PGPATCH infers the `bool pre_arm_checks()` function as the patch location. If terms of a PPL formula's post-conditions are mapped to variables instead of a function, PGPATCH considers functions in which the mapped variables are used as patch location candidates.

To generate the patch, PGPATCH first switches terminal nodes of the expression tree to the found access patterns, as shown in Figure 7. It then generates a patch based on the updated expression tree, conducting an in-order traversal of the expression tree to create an 'if statement". For example, it creates the "if statement" shown in Listing 6 from the expression tree in Figure 7. We note that PGPATCH inserts

[3]ArduPilot provides mission plans for all RV types, which can be used by other RV control programs if they follow the MAVLink protocol [53].

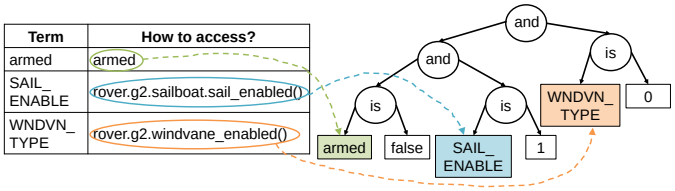


Fig. 7: Illustration of generating the patch for ADD patch type.

```
bool AP_Arming_Rover::pre_arm_checks(...) {
if (armed == false && rover.g2.sailboat.sail_enabled() == 1
    && rover.g2.windvane.enable() == 0) {
        return false; }
```

Listing 6: Fixing the fail-safe bug in sailboat (Section III-B).

the formula's post-conditions (i.e., actions to be executed) into the basic block of the created "if statement". For example, it inserts *return false* to the basic block of the "if statement" as the formula's post-condition (`pre_arm_checks` is `false`) explicitly denotes returning *false* in `pre_arm_checks` function.

*3) Reusing an Existing Code Snippet (*REUSE*):* Logic bugs in REUSE patch type cause an RV control program not to perform a behavior after a specific flight stage although the RV's states satisfy preconditions to activate the behavior. This means that the program already has a code snippet to trigger the correct behavior, but it stops activating the behavior after a specific flight stage. Thus, PGPATCH reuses the existing "if statement" code block to activate the desired behavior in all flight stages. For example, the PPL formula "If `ANGLE_MAX` is less than 1000 or `ANGLE_MAX` is more than 8000, then `prearm` is `error`" defines that `ANGLE_MAX` configuration parameter in ArduPilot must have a valid range from 1K to 8K. However, a logic bug occurs because ArduPilot calls the following function to check whether the `ANGLE_MAX` parameter has a valid value only before the arming stage.

```
bool AP_Arming_copter::parameter_checks() {
  if (angle_max < 1000 || angle_max > 8000) return false; }
```

In these cases, PGPATCH reuses a code snippet that already exists in the source code to generate the patch. Specifically, PGPATCH first finds the existing code snippet, and then inserts the identified code block into a control loop to make the RV execute the behavior at all flight stages. To detail, PGPATCH first extracts all terms from the violated formula and maps the terms into variables in the source code, e.g., *ANGLE_MAX* is mapped to the `angle_max` variable. PGPATCH also extracts the formulas' constant and Boolean values, e.g., it obtains the following terms: *angle_max*, *1000*, and *8000* from the ANGLE_MAX policy. PGPATCH next creates the LLVM *bitcode* from the source code, finds all functions, including an "if statement" (i.e., "cmp" instruction), and obtains def-use chains of all the "cmp" instructions and their operands from these functions. PGPATCH finds an "if statement" that consists of the obtained terms (*angle_max*, *1000* and *8000*).

For instance, in the example shown in Figure 8, PGPATCH performs the following, it finds a def-use chain of `%cmp29` instruction (❶), matches one of the obtained terms (i.e., `8000`) with the operand (i.e., the constant value stored in `0x3110000`) (❷), backtracks the def-use chain of another operand in the

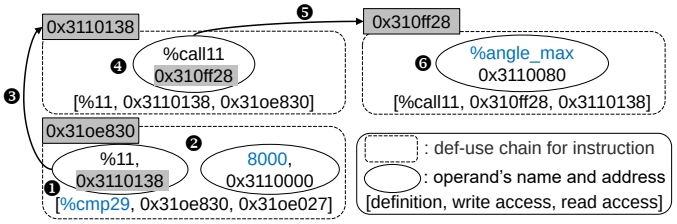


Fig. 8: Backtracking the def-use chains of a `%cmp` instruction.

```
void Copter::one_hz_loop() {
  if (copter.aparm.angle_max < 1000
    || copter.aparm.angle_max > 8000) // raise an error
```

Listing 7: Fixing the `ANGLE_MAX` bug.

`%cmp29` instruction (`%11` in `0x3110138`) (❸), matches the obtained terms with the operand `%call11` (❹), backtracks an operand's def-use chain `%call11` (❺), matches one of the obtained terms (i.e., *angle_max*) with the operand `%angle_max` (❻), and returns the function that includes `%cmp29` because PGPATCH identifies the target "if statement". Thereafter PGPATCH finds the "if statement" in the `parameter_checks()` and it inserts the identified code snippet into a control loop, which is called periodically. We note that PGPATCH requires users to designate the control loop.

Lastly, PGPATCH changes the access patterns for variables and functions in the found code snippet because the patch location is in a different class from the found code snippet. For example, PGPATCH extracts access patterns from the access pattern mapping table and switches `angle_max` to `copter.aparm.angle_max`, as shown in Listing 7. We note that PGPATCH also changes the `'return false;'` statement because the function at the patch location is *type void*. In such a case, PGPATCH prints an error message as a PPL formula's post-condition (i.e., an action to be executed).

*4) Checking Valid Ranges of Parameters (*CHECK*):* To detect and prevent the parameter values outside their valid range, we use the PPL formulas with the following structure. "$\texttt{Min}_{\texttt{i}}$ is less than $\texttt{Param}_{\texttt{i\_value}}$ and $\texttt{Max}_{\texttt{i}}$ is more than $\texttt{Param}_{\texttt{i\_value}}$", where `i` denotes the `i`-th configuration parameter. To fix this type of logic bugs, PGPATCH's patch forces the parameter to have a valid value before a code statement refers to it. PGPATCH performs the following steps: (1) It obtains the name of the configuration parameter triggering a logic bug from the given user inputs. (2) It maps the parameter name to a variable in the source code. (3) It finds all code statements which refer to the identified variable. (4) It learns how to access the identified variable from the access pattern mapping table (Figure 6). (5) It inserts an "if statement" checking for the parameter's value. PGPATCH assigns the default parameter value to the identified variable if it has a value outside the valid range.

PGPATCH obtains the valid ranges and default values of configuration parameters by parsing the XML files in ArduPilot [7], PX4 [65], and Paparazzi [58]. For example, in GPS Fail-Safe Bug (Section III-A), `COM_POS_FS_DELAY` configuration parameter's valid range in PX4 is between 1 and 100. PGPATCH first maps the parameter to the `_param_com_pos_fs_delay` variable. PGPATCH then inserts an "if statement" before

```
bool Commander::check_posvel_validity (...) {
  if (_param_com_pos_fs_delay.get() < 1 ||
      _param_com_pos_fs_delay.get() > 100) {
    // Assign a default value
    _param_com_pos_fs_delay = 1; }
```

Listing 8: Fixing the GPS fail-safe bug (Section III-A).

```
void Copter::failsafe_battery_event(void) {
  else if (g.failsafe_battery_enabled == 2
    && home_distance > 2)  // Switch to RTL
```

Listing 9: Fixing the battery fail-safe bug (Section III-C).

any statement referring to the `_param_com_pos_fs_delay` variable, as shown in Listing 8.

5) Updating a Statement (`UPDATE`*):* RV software must trigger post-conditions only when an RV's current states satisfy preconditions. However, logic bugs in `UPDATE` patch type trigger a behavior although the RV satisfies only a part of the preconditions. This means that the RV software uses an incorrect "if statement". Hence, PGPATCH replaces the incorrect "if statement" with a new one.

PGPATCH first extracts the terms of the violated formula's preconditions. For example, in Battery Fail-Safe Bug (Section III-C), preconditions are "*Failsafe is on*", "*FS_BATT_ENABLE is 2*", and "*home_distance* $>$ *2*". It obtains *Failsafe*, *FS_BATT_ENABLE*, and *home_distance* as the terms. Second, it extracts the variables and functions from the term-source code mapping table (Figure 4), e.g., *FS_BATT_ENABLE* and *home_distance* are mapped to `failsafe_battery_enabled` and `home_distance` variables, respectively. Third, it finds an "if statement" that uses the mapped variables as operands. For this, it backtracks def-use chains of `%cmp` instructions of the unpatched program's *bitcode*. The backtracking method is the same as Section V-D3. Fourth, from the access pattern mapping table, it learns how to access the mapped variables in the found patch location, e.g., we can use `g.failsafe_battery_enabled` to access the *FS_BATT_ENABLE* term in the found patch location (`failsafe_battery_event` function). Lastly, it conducts in-order traversal of the expression tree to create a new "if statement". It then replaces the incorrect "if statement" with the new one (Listing 9). Yet, this patch cannot fix Battery Fail-safe Bug. We will detail how PGPATCH' Patch Verifier corrects the patch in Section V-E.

6) Disabling a Statement (`DISABLE`*):* Logic bugs in `DISABLE` patch type cause an RV software to change a state although a PPL formula explicitly represents the state that must not be changed. It means that the RV software has unnecessary code statements that change the state. Hence, PGPATCH finds the statements and comments them out to fix these logic bugs. PGPATCH first extracts terms from the formula's post-condition part, e.g., in Tilt Angle Bug (Section III-D), "`_constraints.tilt` is `disabled`" is the violated formula's post-condition. It obtains the *_constraints.tilt* term from the formula. Second, it obtains mapped variables from the term-source code mapping table, e.g., *_constraints.tilt* term is mapped to `_constraints.tilt`. Third, it finds all statements that assign a value to the mapped variables, e.g., `_constraints.tilt = a value`. Lastly, it comments out the found statements. PGPATCH leaves a variable declaration and disables only an assignment part if a found statement includes declaration and assignment at the same time, e.g., it comments out statements at lines 2 and 5 in Listing 5.

E. Patch Verifier

PGPATCH checks whether the patch fixes the logic bug for the given user inputs that trigger the bug and performs patch correction if needed. It then verifies (1) the bug does not occur in other missions and environmental conditions (i.e., testing the patches in different contexts), and (2) the patch does not break an RV's functionality and degrade its performance.

1) Patch Correction: We noticed some patches could not fix the bugs due to inconsistencies between metric units used in PPL formulas and the units in the source code. Thus, we developed a patch correction component to address these issues. Patch correction first checks whether the bug persists on the RV simulator with the given user inputs after the patch is deployed. If PGPATCH still detects the bug while executing the bug-triggering inputs on an RV simulator, PGPATCH tries to fix the generated patch by fixing the unit inconsistencies.

Fixing Unit Inconsistency. PGPATCH's patch generation is based on PPL formulas, which do not include the units for the constants. When PGPATCH generates a patch, it makes a guess on the unit of the constants (e.g., meter or centimeter). If the guess is incorrect, PGPATCH attempts to fix the generated patch. Specifically, when the term's name implies a distance variable (e.g., altitude, height, and elevation), PGPATCH converts the distance variable into another unit (i.e., multiplies or divides it by the powers of 10). It then tests whether the bug is fixed with the selected unit.

To illustrate, in the Battery Fail-safe Bug, PGPATCH creates a patch (Listing 9). This patch does not prevent the targeted logic bug as `home_distance` uses centimeters, while the constant number 2 in the formula is in meters. After the patch correction step, PGPATCH generates the following patch:

```
else if (g.failsafe_battery_enabled == 2
        && (home_distance/100) > 2)
```

2) Testing the Completeness of Patches: We consider that a patch is complete if the following three conditions are satisfied: (1) The inputs that previously triggered the bug do not lead to the bug in the patched program (C_1). (2) The patched program does not lead to the bug even in different contexts, e.g., different missions and altitudes (C_2). (3) The patch does not break an RV's existing functionalities and degrade performance (C_3).

To verify these conditions, PGPATCH runs the patched program on an RV simulator using multiple scenarios. Particularly, to check C_1 and C_2, we leverage the "Autotest suite" [12] from ArduPilot and extend it to PX4 and Paparazzi. ArduPilot developers use the test suite to test functionalities after they update their RV software. Whenever the "Autotest suite" tests each scenario, PGPATCH executes the inputs given by users that trigger the logic bug. If PGPATCH does not detect any logic bug, it runs PGFuzz [41] to find a new input set to trigger the logic bug. If PGPATCH detects a logic bug, this means the generated patch does not fully fix the bug.

To verify C_3, PGPATCH checks if the patched program violates any existing formulas obtained from PGFuzz while running PGFuzz for checking C_1 and C_2. Additionally, it runs the "Autotest suite" on the unpatched and patched programs. It then compares the RV's physical states (e.g., roll, pitch, and yaw) from each execution. We obtain the states from stored log files after executing the "Autotest suite". We note that a correct patch may change some states since it fixes a bug. Thus, PGPATCH's Patch Verifier must know which states are expected to be changed by the patch. To address this, we use PGFuzz's profiling engine [41] that automatically finds states related to a given formula ($State_C$). If PGPATCH detects either a policy violation or an unexpected changed state that is not included in $State_C$, we consider that the patch interferes with the program's functionality and performance.

VI. IMPLEMENTATION

Preprocessor. We write 376 lines of code (LoC) in Python using the PyParsing v.2.4.7 library [70] to implement the translator which converts (*i*) LTL formulas to PPL formulas and (*ii*) PPL formulas to LTL formulas. The syntax analyzer (❷ in Figure 1) is implemented in 279 LoC in Python. To create (*i*) the term classification table (❸b Figure 1) and (*ii*) candidate terms for users, we obtain variable and function names in the flight control programs using LLVM v.10.0.0 [50]. We extract the variable and function names through an LLVM pass that consists of 217 LoC in C. We manually construct a list of RV physical states from RVs' documentations [11], [59], [68].

Patch Type Analyzer. We write 429 LoC in Python to implement the "Terms and source code mapper" and 741 LoC in Python for "Patch type analyzer" components (❺-❻ in Figure 1) on top of the Pymavlink v2.4.9 and PPRZLINK libraries [63], [69]. These libraries enable PGPATCH to communicate with a simulated vehicle through MAVLink [53] commands. To simulate RVs, we choose Software in the Loop (SITL) [6] for ArduPilot, jMAVSim [36] for PX4, and NPS [57] for Paparazzi.

Patch Generator. The patch locator is implemented as 239 LoC in Python. We write 457 LoC in C to collect def-use chains of variables through an LLVM pass. To parse XML files which contain valid ranges and default values of configuration parameters, we write 95 LoC in Python.

Patch Verifier. To deploy patches to RV software, we write 114 LoC in Python. To test the patches' completeness, we adapt the "Autotest suite" of ArduPilot v.4.0.3 [12]. It consists of four scripts in Python for each RV type. The "Autotest suite" consists of 4,911 LoC for the multi-copters, 5,435 LoC for the rover, 1,982 LoC for the fixed wings, and 729 LoC for the submarines. To run the "Autotest suite" on PX4, we modify 295 LoC for the multi-copter, 279 LoC for the rover, and 149 LoC for the fixed wing as they differently implement the MAVLink protocol. Further, to run the "Autotest suite" on Paparazzi, we have to modify 1,086 LoC since Paparazzi uses PPRZLINK instead of Pymavlink. Finally, we manually created a list of distance variables to fix unit inconsistencies.

	Selected bugs	Patchable bugs	Fixed bugs
ArduPilot (A)	70	38	32
PX4 (PX)	70	27	24
Paparazzi (PP)	70	29	21
Total	210	94	77

TABLE I: Details of the quantitative evaluation for bugs from the commit history of ArduPilot, PX4, and Paparazzi.

	Bug origin	Fuzzing			Commit history			Total
	RV SW	A	PX	PP	A	PX	PP	
Patch type of fixed bugs	ADD	1	0	0	13	6	14	34
	REUSE	44	0	0	0	0	0	44
	UPDATE	0	0	0	15	18	4	37
	DISABLE	1	0	0	4	0	3	8
	CHECK	94	24	17	0	0	0	135
Unfixable		10	12	0	6	3	8	39
Total		150	36	17	38	27	29	297
Success rate		93.3%	66.7%	100%	84.2%	88.9%	72.4%	86.9%

TABLE II: Summary of the quantitative evaluation on ArduPilot (A), PX4 (PX), and Paparazzi (PP).

VII. EVALUATION

A. Experiment Setup

We evaluate PGPATCH on the three most popular flight control software, ArduPilot, PX4, and Paparazzi. We collect a total of 2,268 logic bugs. In particular, we find 292 logic bugs reported by previous RV fuzzing research papers [41], [43]. We refer to these logic bugs as $DataSet_F$. Further, we collect 1,976 logic bugs by searching in the GitHub commit history of the three considered RV control programs [8], [59], [64]. We refer to these logic bugs as $DataSet_H$.

Among these bugs, we select bugs that satisfy the following criteria: This is because outdated bugs cannot be reproduced on the same version of the operating system and RV simulator. (1) They are reported within the last two years because outdated bugs cannot be reproduced on the same version of the operating system and RV simulator. (2) They can be triggered by sending user inputs to a simulated vehicle. (3) They belong to one of the five patch types that PGPATCH supports.

We select a total of 297 logic bugs from $DataSet_F$ and $DataSet_H$. Specifically, by applying these filtering rules, we obtain 203 logic bugs from $DataSet_F$. Most of the bugs are classified as CHECK (74.6%) and REUSE (24.3%). Further, to choose logic bugs from $DataSet_H$, we first randomly select 210 bugs, then, we obtain 94 (out of 210) logic bugs by applying the above filtering rules, as shown in Table I. To fix bugs from $DataSet_F$, we reuse 29 LTL formulas from PGFuzz [41]. Additionally, we create 94 PPL formulas[4] ourselves to fix logic bugs from $DataSet_H$. Out of these 123 formulas, 4 formulas fixed multiple bugs, while the others can fix one bug each (as detailed in Appendix G).

We run our evaluation using a desktop machine with an Intel i5-10400 CPU, 64 GB RAM, and Ubuntu 18.04 64-bit.

B. Quantitative Evaluation

After the patch passes the patch verification process in PGPATCH's patch verifier, we manually examine the patch to check its correctness. When PGPATCH aborts patch generation

[4]These 94 rules are available online: https://github.com/purseclab/PGPatch/blob/main/policy_list.pdf

Bug origin	Fuzzing			Commit history		
	A	PX	PP	A	PX	PP
Fixed bugs	140	24	17	32	24	21
Performance damage	0	0	0	0	0	0
Different from developers' patches	N/A	N/A	N/A	2	0	0
Total	181			77		

TABLE III: Summary of the qualitative evaluation.

or incorrectly generates a patch, we also manually analyze it to identify what makes PGPATCH fail.

As shown in Table II, PGPATCH correctly fixes 258 out of 297 logic bugs (86.9%). Specifically, PGPATCH's Patch Generator component initially fixes 238 out of 297 logic bugs. PGPATCH's Patch Verifier then detects 59 faulty patches and corrects 20 of them by fixing unit inconsistencies.

PGPATCH fails to fix 39 logic bugs (Table II). Yet, we note that failing to create correct patches does not mean that PGPATCH deploys faulty patches. The false positives represent the patches that PGPATCH considers correct, but they are actually faulty (do not fix the bugs). Overall, PGPATCH produces zero false positives because PGPATCH's Patch Verifier checks the correctness of the patches created by PGPATCH's Patch Generator before they are deployed. We found 39 logic bugs to be faulty for the following reasons: (1) For four patches, PGPATCH should have added third party libraries to compute variables related to attitude control, but using third party libraries is not supported, e.g., a patch [4] needs to add a math library to apply a filter to the *yaw speed* state, (2) 23 patches require implementing a new feature from scratch, e.g., a GitHub commit [2] adds a new flight mode for Quad-plane type drones, (3) For eight patches, PGPATCH needs to define a new variable, loop, or function, e.g., [3], and (4) Four patches require other complicated techniques, e.g., a GitHub commit [78] adds/updates mathematical formulas. Creating correct patches in these cases requires different kinds of analyses that we plan to study in the future.

Patch Creation and Testing Overhead. PGPATCH takes on average 12.5 minutes to create a patch. The Patch Verifier then runs the "Autotest suite" [12] and PGFuzz [41] to verify a patch. These steps take an average of 2.81 hours. In the case of manual patching, the developers similarly leverage the "Autotest suite". They additionally conduct a manual code inspection with other maintainers of RV software.

C. Qualitative Evaluation

To evaluate the correctness of patches generated by PGPATCH, two authors of this paper manually examined each patch. To determine if a patch is correct, we use the following criteria: (1) When both authors agree that a patch created by PGPATCH and a patch created by developers are semantically the same, we consider that the patch created by PGPATCH is correct. (2) Some developers' patches contain supplementary code lines, e.g., logging an RV's states. We ignore such code lines in developers' patches. We believe that developers can easily add such supplementary code to patches created by PGPATCH. (3) To measure patches' performance impact, we compare performance between an unpatched version of the RV software and patched software containing all patches created by PGPATCH.

Patch Correctness. PGPATCH generates 181 correct patches out of 203 bugs found by fuzzing works ($\texttt{DataSet}_\texttt{F}$) [41], [43], as shown in Table III. Specifically, the root cause of 179 out of 181 bugs is that the three RV programs do not check valid ranges of parameters or use incorrect ranges. To fix these bugs, PGPATCH extracted valid ranges for the configuration parameters from the RV documentation in XML file format, as explained in Section V-D4. For these bugs, the developers updated the documentation to warn users not to assign parameter values outside valid ranges, and, unfortunately, have not yet included patches for them at the time of writing.

PGPATCH generates 77 correct patches for logic bugs found from $\texttt{DataSet}_\texttt{H}$. We found that 97.4% (75/77) of the patches are semantically the same as developers' patches. However, two patches of the DISABLE patch type are semantically different from developers' patches. The reason for the difference is that PGPATCH is not designed to remove existing variables and functions while using the DISABLE patch type. We select such a design choice to minimize damaging the functionalities of RV programs. However, developers' patches remove functions. We note that the two patches created by PGPATCH still fix the logic bugs. However, PGPATCH's DISABLE patch type might generate patches that contain unused code snippets.

Performance Impact. We evaluate the performance impact of the 258 patches. After deploying all the generated patches simultaneously (172 patches for ArduPilot, 48 patches for PX4, and 38 patches for Paparazzi), we do not observe any significant performance degradation.

D. Root Cause and Physical Effect of Bugs

The root causes of the 181 bugs reported by fuzzing tools for RVs [41], [43] are mainly incorrectly checked valid ranges of configuration parameters (96.1%) because the fuzzers heavily mutated the parameters and discovered that RV control programs do not properly check valid ranges of the parameters. Out of 181 bugs, 176 bugs (97.2%) directly lead to physical harm (i.e., either crashing or instability), and 5 bugs (2.8%) cause incorrect states. In contrast, we found that 66 out of 77 bugs (85.7%) from GitHub commit history occur due to either developers' mistakes or unimplemented features. Out of 77 bugs, 46 of them (59.7%) directly cause physical harm. 31 (40.3%) either degrade flight performance due to increased processing time, wasted memory space, and incorrectly measured states (e.g., incorrect land detection) or lead to incorrect states. We detail the root cause and physical effects of bugs for each RV control program in Appendix C.

E. Generality of PGPATCH

To evaluate the generality of PGPATCH to other RV software, we studied 11 open-source RV control programs commonly used in research [24] and industry [21], [22], as shown in Table IV. To port PGPATCH to other RV software, the following conditions must be satisfied: (1) The RV software uses a telemetry protocol (e.g., MAVLink [53]) between an RV and a ground control station (GCS) so that PGPATCH can

RV software	Language	SLOC (K)	Telemetry protocol	Simulation	Naming convention	Portable
Paparazzi [59]	C	6,024	✓	✓	N/A	✓
PX4 [64]	C/C++	4,042	✓	✓	✓	✓
ArduPilot [8]	C/C++	3,999	✓	✓	✓	✓
Betaflight [14]	C	1,825	✓	✓	N/A	✓
Cleanflight [18]	C	1,668	✓	✓	N/A	✓
INAV [35]	C	1,309	✓	✗	N/A	✗
LibrePilot [47]	C	1,106	✓	✓	N/A	✓
Tau Labs [76]	C/C++	944	✓	✓	✓	✓
dRonin [23]	C/C++	817	✓	✓	✓	✓
Hackflight [29]	C++	34	✗	✓	✓	✗
multiwii [55]	C++	13	✓	✗	✗	✗

TABLE IV: A list of RV control programs used to evaluate the generality of PGPATCH.

operate the RV through the telemetry protocol. (2) A simulator supports the RV software. (3) The RV control program is implemented in C/C++. (4) The software follows naming conventions of getters and setters, e.g., *get*, *set*, *enabled*, and *disabled*, if the software is implemented in C++. We manually analyzed the documentation and source code of RVs, and executed each RV software to test the first three conditions (1)-(3). To verify (4), we randomly select private member variables and check whether the RV software follows the naming conventions (See Appendix D). We found that PGPATCH can be ported to 8 of the 11 RV control programs with minor engineering effort. However, PGPATCH cannot be easily applied to INAV [35] and multiwii [55], and Hackflight [29] as they either are not supported by any simulators or the telemetry protocol is used for viewers instead of operating RVs.

F. Discovering and Fixing New Logic Bugs

PGPATCH, when used in conjunction with bug-discovery tools (such as PGFuzz [41]), can help identify and fix new bugs. In particular, we used PGPATCH together with PGFuzz to leverage formulas for discovering new bugs and fixing them. To demonstrate this use case scenario, we evaluate PGPATCH on Paparazzi to find new logic bugs. We reused the five formulas of PGFuzz and created an additional PPL formula that ensures the RV's correct operation under fail-safe mode. PGPATCH, in conjunction with PGFuzz, ran for a day, and discovered a total of 12 previously unknown logic bugs (See Table VIII in Appendix E). PGPATCH fixed 6 logic bugs correctly, 4 `ADD`, 1 `DISABLE`, and 1 `CHECK` type. The other 6 bugs require creating the features from scratch (e.g., requires implementing `Hover` flight mode and `Quad_Elie0` and `Quad-Navstik` vehicle types). We have reported the discovered, previously unknown 12 bugs to Paparazzi developers, and they acknowledged all the bugs.

G. User Study

In our user study, we aim to determine (1) the effort required to create PPL formulas, and (2) how useful PGPATCH is in patching the logic bugs compared to the manual patching effort required by RV developers and users.

Recruitment Methods. We recruit from two different groups, (1) RV software developers who actively fix reported bugs, and (2) experienced RV users who create patches, which are then reviewed and approved by developers before they are applied to the RV software. 22 participants (6 developers and 16 users) applied for our study, and 18 (6 developers and 12 users) qualified based on their experience in finding bugs in RVs and modifying RV software. We note that 1 of the 6 RV developers is an official maintainer of ArduPilot. We detail the demographic data of the participants in Appendix F.

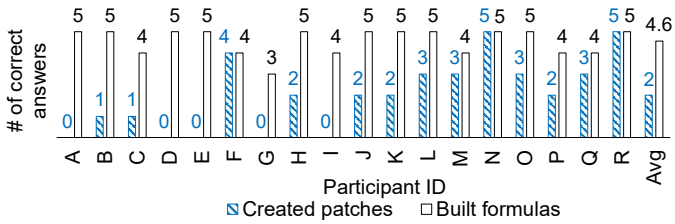


Fig. 9: The number of correct answers per participant.

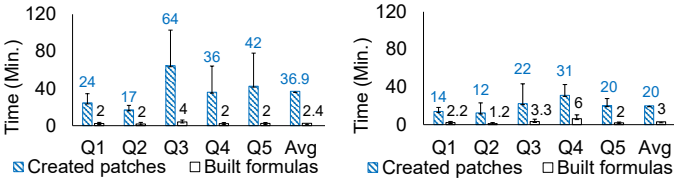
	% of correct answers	# of incorrect answers	Reasons	Type of fault
Patch	80.4% 37/46	9	Partially fix a bug (5)	Semantic
			Wrong unit used (2)	
			Wrong patch location (1)	
			Compile error (1)	Syntactic
Formula	91.1% 82/90	8	Wrong term (3)	Syntactic
			Wrong verb (2)	
			Wrong value (1)	
			Missing post-condition (2)	Semantic

TABLE V: Root causes of incorrect answers.

1) Tasks: The user study consists of two tasks: (*i*) patching five logic bugs in ArduPilot at the source code level, and (*ii*) creating five PPL formulas used by PGPATCH to fix the five logic bugs in ArduPilot. We randomly selected the five bugs from our evaluation data set (Section VII-A). We randomized the order of the tasks and bugs per participant. We limit the maximum time to perform each task to 2 hours.

Task1: Manually Creating Patches. Task1 consists of five questions. In each question, the participants are asked to read a logic bug's description and manually create a patch. The description explains (1) the incorrect behavior caused by the bug, (2) the correct behavior on the documentation, and (3) how to trigger the bug on the simulator. We ask the participants to submit the patch locations and source code files containing their patches. After each question, the participants rate (*i*) their confidence in their patch, and (*ii*) the difficulty in fixing the logic bug on a five-point scale. We allow participants to give up creating a patch and justify it by providing an explanation.

Task2: Building PPL Formulas. In Task2, we ask participants to create five PPL formulas. We show an online self-tutorial to participants that (1) explains the PPL syntax in Listing 1, (2) provides two PPL templates in Section II, and (3) presents six examples to demonstrate how to convert the descriptions of RV behaviors into PPL formulas. This is because the participants have experience with RV software but not with the PPL syntax. The participants spent on average 8.4 minutes (min: 4.9 and max: 11.8 mins) in the tutorial. In each question, we ask the participants to read a description of the RV's correct behavior and create a PPL formula using a set of candidate terms. We provide these candidate terms to participants since PGPATCH outputs them to allow users to build formulas (Section II). After each question, we ask the participants to (*i*) score their confidence in their formula and how difficult it was to create it, and (*ii*) explain any reason if they decide to give up.



(a) Experienced RV users. (b) Experienced RV developers.

Fig. 10: The spent time per question.

(a) Experienced RV users. (b) Experienced RV developers.

Fig. 11: Confidence and difficulty on a five-point scale.

2) User Study Results: We compare the manual patches and PPL formulas created by participants using four metrics: (1) the number of correct answers, (2) average time spent in each question, (3) the participants' confidence in their answers, and (4) the difficulty level participants assign. The participants create on average 4.6 correct PPL formulas compared to 2 correct manual patches while spending about 12 times less time on creating formulas. Additionally, the participants have higher confidence in PPL formulas compared to patches (on average 4.2 vs. 3) and they find generating PPL formulas easier (on average 1.9 vs. 3.3). We confirm the differences are statistically significant using the Mann-Whitney U test [51].

Correctness. Two authors of this paper met over multiple sessions to check the correctness of participants' answers (manual patches and PPL formulas) and reconcile disagreements. Through this analysis, we categorize each answer as 'correct', 'incorrect' or 'incomplete' (empty answer).

As shown in Figure 9, the participants correctly created on average 2 manual patches and 4.6 PPL formulas. We found that the difference between the two averages is statistically significant, with $p < 0.001$. We next analyzed the correctness of users' and developers' answers separately. In Figure 9, participants from `A` to `L` are experienced RV users and participants from `M` to `R` are RV software developers. We found that RV users correctly created on average 1.25 patches and 4.6 formulas, while RV developers correctly created on average 3.5 patches and 4.5 formulas. We found the difference between the number of correct PPL formulas from users and developers is not statistically significant ($p = 0.66$), whereas the difference between the number of correct patches from users and developers is ($p = 0.008$). This shows that participants can correctly write PPL formulas and create patches through PGPATCH regardless of their level of RV software experience.

We examined the reasons behind the incomplete and incorrect answers from participants. We found that none of the participants provided an incomplete PPL formula, but they did not give an answer for 44 of the 90 manual patches. We observed "limited time" and "not familiar with this component" were the main reasons participants provided in free text for giving up creating patches. Among the 46 manual patches and 90 PPL formulas participants provided, we found participants build PPL formulas more accurately compared to creating code-level patches. Specifically, 19.6% (9/46) of the patches and 8.9% (8/90) of the PPL formulas were incorrect (See Table V). Upon further analysis, we found the incorrect manual patches stem from (1) partially fixing a bug, (2) using a wrong unit (i.e., meter or centimeter), (3) wrong patch location, and (4) introducing compilation errors. Regarding incorrect PPL formulas written by participants, they were in almost all cases (6 out of 8) syntactically incorrect formulas ("wrong verb", "wrong term", and "wrong value"). PGPATCH can detect the syntactically incorrect formulas (Section V-B) and avoid using them to generate a patch.

Required Time. We measured the time participants spent answering each question. We exclude the time the participant spent on a question if its answer is incomplete. We found participants spent on average 31.7 minutes on each manual patch and 2.6 minutes on each PPL formula. We found the difference between the average time spent on manual patches and PPL formulas is statistically significant ($p < 0.001$). As shown in Figure 10a, the experienced RV users spent on average 36.9 and 2.4 minutes on creating a patch and building a formula, respectively. As shown in Figure 10b, RV developers spent on average 20 minutes to create each patch and 3 minutes to build each formula. We found the differences between the average time users spent on patches and formulas ($p < 0.001$) and the average time developers spent on patches and formulas ($p < 0.001$) are both statistically significant. These results show that building formulas requires less time regardless of the level of experience in RV software development.

Confidence and Difficulty Scores. We asked the participants to specify, for each question, their levels of confidence and the difficulty of the question on a scale from 1 to 5. The participants gave on average 4.2 confidence level to PPL formulas and 3 to patches they created. We found this difference statistically significant ($p = 0.019$). The participants rated the difficulty as 1.9 for PPL formulas and 3.3 for patches on average ($p < 0.001$). Based on these, we conclude that participants have a higher confidence in PPL formulas, and they find generating manual patches more difficult. As shown in Figure 11a, experienced RV users felt high confidence and less difficulty when they built formulas compared to creating patches. Although the RV developers also felt on average higher confidence and less difficulty in building formulas (See Figure 11b), the difference between creating patches and building formulas is smaller compared to the users' answers.

In summary, our user study shows that, regardless of the participant's experience level, creating PPL formulas to fix bugs using PGPATCH is both easier and less error-prone than manually fixing the source code of an RV software package.

VIII. Related Work

A. Test suite-based Automatic Program Repair

Search-based APR. A line of search-based APR methods mutates a statement's operation at a candidate patch location [20]. However, changing an operation (e.g., arithmetic and relational)

could not fix any bugs in RV software (See Section III). Other works use existing code to fix bugs rather than synthesizing a new code snippet [27], [79], [80]. While reusing code can fix some of the logic bugs (See Section VII-B), patching logic bugs requires more complex analysis, e.g., updating a conditional statement, and inserting a new statement.

Pattern-based APR. To find the patch location and patch the bug, pattern-based APR methods use common "fix patterns" learned from patches written by developers and through static analysis [40], [49]. However, these methods cannot create a patch if the patch needs to access a variable or function outside the patch location. PGPATCH addresses this issue by creating an access pattern mapping table that represents how to access required variables/functions from the patch location. Further, these methods fail to create a patch requiring multiple statements, as they can only fix a single statement. PGPATCH supports patches with multiple statements by synthesizing a patch from a PPL formula.

Satisfiability Modulo Theories (SMT) Solver-based APR. SMT solver-based approaches conduct the following steps to fix bugs [54], [56]. (1) They extract a repair constraint based on symbolic execution. (2) They generate potential patches by enumerating all the possible expressions that can be constructed starting from a set of program variables and operators. (3) They query an SMT solver to check whether a potential patch makes the program pass the given test suite. However, these approaches fail to create correct patches in RV software because the correctness of patches depends on the test suite's completeness and achieving completeness in RV software is challenging due to the huge input/output space.

B. Specification-based Automatic Program Repair

APR on Source Code. Recent works leveraged safety specifications to find patch locations and create code-level patches [34]. For example, they define a memory safety specification for buffer overflow that states "the program must not make an out-of-bounds memory access to a buffer" [48]. However, these methods cannot fix logic bugs in RVs for two main reasons. (1) Their safety specifications solely express memory-safety violations. (2) They insert an "if check statement" as a patch to prevent access to the buffer's out-of-bounds memory. However, fixing logic bugs requires more complex code modifications (See Section III).

APR with Abstract Behavioral Models. A line of work represents the code in an abstract behavioral model (e.g., discrete state transition system) and uses temporal logic formulas to detect and repair software bugs [5], [15], [37]. For instance, AutoTap [82] takes an LTL formula and fixes a bug in trigger-action programming (TAP) rules for IoT devices. This approach has two fundamental limitations preventing it from patching bugs in RV software. First, it assumes that a device command is always executed when the IoT system satisfies the preconditions to trigger it. However, this assumption does not hold in RVs as logic bugs could make the RV software fail to trigger a behavior or execute an action. Second, it uses model checking on a finite Buchi Automaton to validate all possible system executions. However, RV states are mostly represented as floating point numbers, which makes extracting behavior models and building the equivalent Buchi Automaton challenging. Additionally, RV software operates both in digital and physical spaces; thus, its behavior can be represented in a hybrid automaton rather than a discrete transition system. This makes model checking undecidable due to infinite state space in hybrid systems [31], [62]. These limitations are also valid for other APR methods that operate on similar abstract behavioral models.

IX. Limitations and Discussion

Correctness of a Given Formula. PGPATCH verifies the syntax and semantics of the given formulas (Section V-B2). Yet, it cannot detect an incorrect PPL formula written by developers, if the formula does not conflict with any other formulas. In such a case, PGPATCH could produce a faulty patch. One possible solution to this problem is automated extraction of policies from the correct function of RVs. Appendix B provides more details on this topic.

Effort to Port PGPATCH to other RV Software. Porting PGPATCH to Paparazzi required the following steps: (1) Defining formulas that describe an RV software's correct behavior by reusing existing formulas and, if necessary, updating the terms on the formulas based on variable/function names provided by PGPATCH; (2) Verifying and updating the term classification table; (3) Writing policy violation predicates according to the new formulas; (4) Designating a control loop function for generating `REUSE` type patches; (5) Updating the list of distance variables for fixing metric unit inconsistency. We believe that these tasks are not a burden for developers familiar with RV control programs. For instance, porting PGPATCH from PX4 to Paparazzi required about 14.5 hours. This includes above manual tasks and modifying 815 LoC in the "Autotest suite" (Section V-E2) to adapt to Paparazzi's differences in the MAVLink protocol.

Opting out of the Patch Type Analyzer. PGPATCH allows the patch type analyzer to be optional for creating patches. To detail, when a user does not select the patch type analyzer to create a patch, PGPATCH's Patch Generator first creates a patch per each patch type that PGPATCH supports, PGPATCH's Patch Verifier then validates whether the created patches fix the bug. It then selects the patch that fixes the bug and deploys that patch, ignoring the other generated patches.

X. Conclusions

We introduce PGPATCH, a policy-guided APR framework for RVs. PGPATCH fixes the logic bugs via customized methods for five patch types. It addresses the unique challenges in patching RV software by using PPL formulas to find patch locations and generate patches. We evaluated PGPATCH on three popular flight control software. PGPATCH correctly fixed 258 out of 297 logic bugs (86.9%) without interfering with the RV's intended functionality and performance. Our user study, involving 18 experienced RV developers and users, shows that using PGPATCH to fix bugs in RV software is easier and less error-prone than manually patching the bugs.

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. This work was also supported in part by DARPA under contract number N6600120C4031. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] "Add arming check for windvane," https://tinyurl.com/2byfhbzd, 2021.
[2] "Add new mode," https://tinyurl.com/swshv3tc, 2021.
[3] "Add vertical emergency braking," https://tinyurl.com/2w5h2eds, 2021.
[4] "Add yaw speed filter," https://tinyurl.com/2rbzczn9, 2021.
[5] D. Alrajeh and R. Craven, "Automated error-detection and repair for compositional software specifications," in *Proceedings of the International Conference on Software Engineering and Formal Methods (SEFM)*, 2014.
[6] "Apm sitl," https://tinyurl.com/wzkamnrp, 2021.
[7] "Ardupilot parameter list xml," https://tinyurl.com/4bemvrh5, 2021.
[8] "Ardupilot project," https://github.com/ArduPilot/ardupilot, 2021.
[9] "Ardupilot blog," https://discuss.ardupilot.org, 2021.
[10] "Ardupilot chat channel," https://ardupilot.org/discord, 2021.
[11] "Ardupilot-documentation," https://ardupilot.org/ardupilot/, 2021.
[12] "Autotest," https://tinyurl.com/6ampumny, 2021.
[13] "Battery fail-safe bug," https://tinyurl.com/ynf788f4, 2021.
[14] "Betaflight," https://github.com/betaflight/betaflight, 2021.
[15] C.-H. Cai, J. Sun, and G. Dobbie, "Automatic b-model repair using model checking and machine learning," *Automated Software Engineering*, 2019.
[16] M. Chen, F. Fischer, N. Meng, X. Wang, and J. Grossklags, "How reliable is the crowdsourced knowledge of security implementation?" in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2019.
[17] H. Choi, S. Kate, Y. Aafer, X. Zhang, and D. Xu, "Cyber-physical inconsistency vulnerability identification for safety checks in robotic vehicles," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.
[18] "Cleanflight," https://github.com/cleanflight/cleanflight, 2021.
[19] "Copy code from stack overflow," https://tinyurl.com/4mf3dhzz, 2021.
[20] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2010.
[21] "Open source drone software projects," https://dojofordrones.com/open-source-drone/, 2021.
[22] "List of flight controller firmware projects," https://blog.dronetrest.com/flight-controller-firmware/, 2021.
[23] "dronin," https://github.com/d-ronin/dRonin, 2021.
[24] E. Ebeid, M. Skriver, K. H. Terkildsen, K. Jensen, and U. P. Schultz, "A survey of open-source uav flight controllers and flight simulators," *Microprocessors and Microsystems*, 2018.
[25] "Ardupilot fence," https://tinyurl.com/3z2w22d9, 2021.
[26] C. Feng, V. R. Palleti, A. Mathur, and D. Chana, "A systematic framework to generate invariants for anomaly detection in industrial control systems." in *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, 2019.
[27] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the conference on Genetic and evolutionary computation (GECCO)*, 2009.
[28] "Gps-failsafe," https://docs.px4.io/master/en/config/safety.html, 2021.
[29] "Hackflight," https://github.com/simondlevy/Hackflight, 2021.
[30] "Hackflight gcs," https://github.com/simondlevy/HackflightGCS, 2021.
[31] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" *Journal of computer and system sciences*, 1998.
[32] H. Huang, S. Z. Guyer, and J. H. Rife, "Detecting semantic bugs in autopilot software by classifying anomalous variables," *Journal of Aerospace Information Systems*, 2020.
[33] Z. Huang, M. DAngelo, D. Miyani, and D. Lie, "Talos: Neutralizing vulnerabilities with security workarounds for rapid response," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2016.
[34] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using safety properties to generate vulnerability patches," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019.
[35] "Inav," https://github.com/iNavFlight/inav, 2021.
[36] "jmavsim," https://github.com/PX4/jMAVSim, 2021.
[37] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program repair as a game," in *Proceedings of the International conference on computer aided verification (CAV)*, 2005.
[38] B. Jobstmann, S. Staber, A. Griesmayer, and R. Bloem, "Finding and fixing faults," *Journal of Computer and System Sciences*, 2012.
[39] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, 2002.
[40] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, 2013.
[41] H. Kim, M. O. Ozmen, A. Bianchi, Z. B. Celik, and D. Xu, "PGFUZZ: Policy-guided fuzzing for robotic vehicles," in *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, 2021.
[42] 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 *Proceedings of the USENIX Security Symposium (USENIX SEC)*, 2020.
[43] 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 *Proceedings of the USENIX Security Symposium (USENIX SEC)*, 2019.
[44] 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.
[45] X. B. D. Le, F. Thung, D. Lo, and C. Le Goues, "Overfitting in semantics-based automated program repair," *Empirical Software Engineering*, 2018.
[46] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," *ACM Sigplan Notices*, 2003.
[47] "Librepilot," https://github.com/librepilot/LibrePilot, 2021.
[48] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie, "Autopag: towards automated software patch generation with source code root cause identification and repair," in *Proceedings of the ACM symposium on Information, computer and communications security (ASIACCS)*, 2007.
[49] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Avatar: Fixing semantic bugs with fix patterns of static analysis violations," in *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019.
[50] "Llvm," https://releases.llvm.org/10.0.0/docs/, 2021.
[51] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, 1947.
[52] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, 2017.
[53] "Mavlink," https://mavlink.io/en, 2021.
[54] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, 2016.
[55] "multiwii," https://github.com/multiwii/multiwii-firmware, 2021.
[56] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, 2013.
[57] "Nps," https://wiki.paparazziuav.org/wiki/NPS, 2021.
[58] "Paparazzi parameter list xml," https://tinyurl.com/np2e6v2r, 2021.
[59] "Paparazzi uas," https://github.com/paparazzi/paparazzi/, 2021.
[60] "Paparazzi chat channel," https://gitter.im/paparazzi/discuss, 2021.
[61] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan *et al.*, "Automatically patching errors in deployed software," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[62] E. Plaku, L. E. Kavraki, and M. Y. Vardi, "Falsification of ltl safety properties in hybrid systems," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2009.
[63] "Pprzlink," https://github.com/paparazzi/pprzlink, 2021.
[64] "Px4 drone autopilot," https://github.com/PX4/PX4-Autopilot, 2021.
[65] "Px4 parameter list xml," https://tinyurl.com/5fhcuydx, 2021.
[66] "Px4 blog," https://discuss.px4.io, 2021.
[67] "Px4 chat channel," https://gitter.im/PX4/Firmware, 2021.
[68] "Px4-documentation," https://docs.px4.io/master/en/, 2021.
[69] "Pymavlink," https://github.com/ArduPilot/pymavlink, 2021.
[70] "Pyparsing," https://github.com/pyparsing/pyparsing, 2021.
[71] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2015.
[72] C. Ragkhitwetsagul, J. Krinke, M. Paixao, G. Bianco, and R. Oliveto, "Toxic code snippets on stack overflow," *IEEE Transactions on Software Engineering (TSE)*, 2019.
[73] N. M. Rodday, R. d. O. Schmidt, and A. Pras, "Exploring security vulnerabilities of unmanned aerial vehicles," in *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2016.
[74] S. Schechter, "Common pitfalls in writing about security and privacy human subjects experiments, and how to avoid them," *Microsoft*, 2013.
[75] M. Shahzad, M. Z. Shafiq, and A. X. Liu, "A large scale exploratory analysis of software vulnerability life cycles," in *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, 2012.
[76] "Tau labs," https://github.com/TauLabs/TauLabs, 2021.
[77] "Tilt-bug," https://tinyurl.com/6yj5bx4v, 2021.
[78] "Update math functions," https://tinyurl.com/eaez8sjn, 2021.
[79] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Communications of the ACM*, 2010.
[80] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, 2009.
[81] 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 *Proceedings of the USENIX Security Symposium (USENIX SEC)*, 2018.
[82] L. Zhang, W. He, J. Martinez, N. Brackenbury, S. Lu, and B. Ur, "Autotap: synthesizing and repairing trigger-action programs using ltl properties," in *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, 2019.
[83] L. Zhang, W. He, O. Morkved, V. Zhao, M. L. Littman, S. Lu, and B. Ur, "Trace2tap: Synthesizing trigger-action programs from traces of behavior," *The ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2020.

APPENDIX

A. Analysis of Bug Types

We collected 1,554 patches (called $\texttt{P}_{\texttt{set}}$) from 2014 to 2021 on their GitHub repositories [8], [64]. Two authors of this paper reviewed and classified each patch into one of three types: (1) fixing a logic bug, (2) patching a memory bug, and (3) minor issues. Here, the minor issues cannot negatively change an RV's behavior, e.g., updating comments and code refactoring. We concluded that $\texttt{P}_{\texttt{set}}$ fixed 1,234 logic bugs, 23 memory bugs, and 297 minor issues. We excluded the 297 minor issues from $\texttt{P}_{\texttt{set}}$ because they are not fixing bugs. Then, we noticed that 98.2% (1,234/1,257) and 1.8% (23/1,257) of bugs are logic and memory bugs, respectively.

B. Automatically Extracting Policies

Previous works have profiled the normal behaviors of (1) programs from values of memory locations [61], and (2) IoT systems and industrial control systems from actuator and sensor traces [26], [83]. One may consider using these methods for profiling the correct behavior of RVs to generate PPL formulas for PGPATCH. Yet, since RVs operate in dynamic outdoor environments, an RV's correct behavior is influenced by many different factors (including user commands and environmental conditions, such as wind, other vehicles, and obstacles). For this reason, profiling complete normal behaviors from the huge input space of RVs is not scalable because it requires testing each user input combination under all possible environmental conditions.

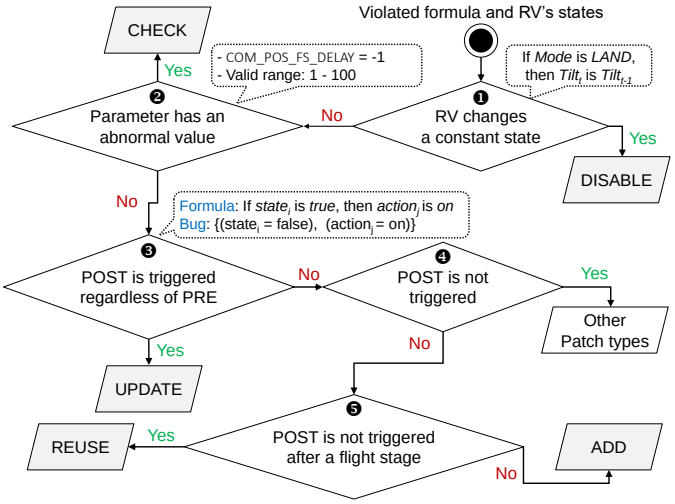


Fig. 12: Flow diagram of PGPATCH' patch type analyzer. *POST* and *PRE* denote post-conditions and preconditions of a PPL formula, respectively.

Algorithm 1 Patch Type Analyzer for ADD and REUSE

Input: A set of default mission plans $\texttt{Mission}_{\texttt{set}}$, a size of $\texttt{Mission}_{\texttt{set}}$ α, a set of inputs to trigger a logic bug $\texttt{Input}_{\texttt{bug}}$, a size of $\texttt{Mission}_{\texttt{set}}$ and $\texttt{Input}_{\texttt{bug}}$ β, a simulator SIM, PPL formulas ϕ
Output: A patch type P

```
1: function INFER_TYPE(Mission_set, α)                          ▷ Main
2:     for i < α; i++ do                          ▷ Mutate a test case Test_set
3:         for j < α; j++ do
4:             if i = j then
5:                 Test_set(j) ← input_bug        ▷ j-th input will be Input_bug
6:             end if
7:             Test_set(j) ← Mission_set(j)       ▷ Assign a next flight state
8:         end for
9:         C ← CHECK_POLICY(Test_set, β)          ▷ Get bug context (C)
10:    end for
11:    P ← PATCH_TYPE(C)                          ▷ Pick a patch type based on C
12:    return P
13: end function
14: function CHECK_POLICY(Test_set, β)
15:    while V = ∅ or k < β do
16:        input ← Test_set(k)                                  ▷ Get an input
17:        S ← SIM.execute(input)          ▷ Collect RV's states (S) from SIM
18:        V ← POLICY_CHECK(ϕ, S)                  ▷ Check policy violations
19:        k = k + 1
20:        if V ≠ ∅ then
21:            return ⟨V, S, k⟩
22:        end if
23:    end while
24: end function
```

RV software	Bug type	# of fixed bugs	Root cause				Physical effect			
			Removed feature	Misimplementation	Unimplementation	Incorrect valid range check	Crash on the ground	Unstable attitude/position	Performance degradation	Incorrect state
ArduPilot	Logic	20	0	4	0	16	7	9	0	4
	Memory	120	0	0	0	120	120	0	0	0
PX4	Logic	24	0	3	0	21	0	23	0	1
	Memory	0	0	0	0	0	0	0	0	0
Paparazzi	Logic	17	0	0	0	17	0	17	0	0
	Memory	0	0	0	0	0	0	0	0	0
Total		181	0	7	0	174	127	49	0	5

TABLE VI: Summary of the fixed 181 bugs that were discovered by PGFuzz.

RV software	Bug type	# of fixed bugs	Root cause				Physical effect			
			Removed feature	Misimplementation	Unimplementation	Incorrect valid range check	Crash on the ground	Unstable attitude/position	Performance degradation	Incorrect state
ArduPilot	Logic	31	2	16	13	0	2	12	5	12
	Memory	1	0	1	0	0	1	0	0	0
PX4	Logic	24	0	20	0	4	1	14	0	9
	Memory	0	0	0	0	0	0	0	0	0
Paparazzi	Logic	19	2	15	0	2	5	9	2	3
	Memory	2	0	1	0	1	2	0	0	0
Total		77	4	53	13	7	11	35	7	24

TABLE VII: Summary of the fixed 77 bugs that are reported in GitHub repositories.

C. Details of the Fixed Bugs

In our evaluation (Section VII), PGPATCH succeeded in patching a total of 258 out of 297 logic bugs. We present the bug types, their root cause, and physical effects in Table VI and Table VII.

Root Causes of Bugs. We group the root causes of the 258 patched bugs into four different categories: (1) "Removed feature" means that an RV control program still can trigger a deprecated behavior, although the documentation explicitly mentions the behavior is prohibited. For instance, `AUTO_DISARMING_DELAY` configuration parameter is deprecated based on the ArduPilot documentation; however, the RV software includes a code snippet for the parameter. This allows attackers to exploit this parameter to disarm an RV's motors (See in Section II). (2) The "Misimplementation" bugs occur when developers incorrectly implement features. For example, in Listing 4, we show a bug that contains an incorrect condition statement, which leads to an improper flight mode change. (3) The "Unimplemented" bugs refer to unimplemented features in the RV software, although the documentation explicitly mentions that the RV software supports the features. For instance, developers forgot to check the validity of a wind vane before the takeoff flight stage (Listing 3), which makes the RV fail to navigate into a waypoint. (4) The "Incorrect valid range check" means that the RV software does not check valid values of the configuration parameters or incorrectly enforces them. For example, in Listing 2, we show that PX4 does not check the valid range of the parameter, which prevents the RV from triggering the GPS fail-safe, and causing it to randomly fly in the air (Detailed in Section III-A).

As shown in Table VI, the root causes of bugs reported by the fuzzing tools [41], [43] are mainly due to the "Incorrect valid range check" (96.1%). This is because the RVs include many configuration parameters, and not properly checking their valid ranges results in logic bugs. As shown in Table VII, 66 out of the 77 bugs (85.7%), which are obtained from GitHub commit history, are due to root causes of "Misimplementation" and "Unimplemented".

Physical Effect of Bugs. We additionally categorize the effects of bugs into four different types: (1) The "Crash on the ground" means that the (simulated) RV loses attitude control and crashes on the ground. (2) The "Unstable attitude/position" refers to instability in either its attitude or position control. (3) The "Performance degradation" represents degraded processing time or wasted memory space. (4) The "Incorrect state" means all other unexpected behaviors, including incorrectly triggered flight mode, failing to detect landing on the ground, and missed warning messages on a ground control station.

We show in Table VI that 176 out of the 181 bugs (97.2%) from PGFuzz directly lead to physical harm (i.e., either crashing or instability), and 5 out of the 181 bugs (2.8%) cause incorrect states. As shown in Table VII, 46 out of the 77 bugs (59.7%) from Github repositories directly cause physical harm. 31 out of the 77 bugs (40.3%) (*i*) degrade flight performance due to increased processing time, wasted memory space, and incorrectly measured states (e.g., incorrect land detection) or (*ii*) lead to incorrect states.

D. Generality of PGPATCH

We randomly select private member variables and check whether the RV software follows the naming conventions. We consider a specific RV software package as complying with the naming conventions if we could find getters and setters for the selected private variables.

The results of our analysis are shown in Table IV. PGPATCH could be ported to the following eight RV control programs with minor engineering effort: Paparazzi [59], PX4 [64], ArduPilot [8], Betaflight [14], Cleanflight [18], LibrePilot [47], Tau Labs [76], and dRonin [23]. We call these eight RV control programs portable RV software packages. On the contrary, PGPATCH cannot be easily applied to INAV [35] and multiwii [55] because they are not supported by any simulators. Further, we cannot deploy PGPATCH to Hackflight [29] because its telemetry protocol [30] is mainly for viewers instead of operating RVs. Particularly, Hackflight's protocol only allows PGPATCH to change *roll*, *pitch*, *yaw*, and *throttle* states of

ID	Vehicle type	Formula ID	Violated formula	Description	Patch type	Fixable?
Bug_1	Mentor Energy	PP.FailSafe	If GPS_{loss} is true and RC_{valid} is false, then mode is FAILSAFE	FAILSAFE mode is not triggered when the vehicle loses GPS and RC signals.	ADD	✓
Bug_2	Minion_Lia					✓
Bug_3	Quad_Lisa_2			FAILSAFE mode is not triggered when the vehicle's mode is $Hover_C$.		✓
Bug_4	Quad_LisaMX			FAILSAFE mode is not triggered when the vehicle's mode is NAV.		✓
Bug_5	Bebop2	PP.Hover	If $Mode_t$ is Hover, then Pos_t is Pos_{t-1} and Yaw_t is Yaw_t	The vehicle fails to stay at a constant position after conducting FLIP mode.	Other	✗
Bug_6	Bebop2			The vehicle fails to trigger Hover mode.		✗
Bug_7	Ardron2					✗
Bug_8	Ardron2			The vehicle crashes on the ground after conducting FLIP mode.		✗
Bug_9	Quad_Elie0	$PP.TAKEOFF_1$	If $Command_t$ is takeoff, then ALT_{target} is greater than or equal to HOME_ALT + 5	The vehicle fails to take off from the ground.		✗
Bug_{10}	Quad-Navstik					✗
Bug_{11}	LadyLisa	$PP.Hover_Z$	If $Mode_t$ is $Hover_Z$ and $Throttle_t$ is 1500, then ALT_t is ALT_{t-1}	When the vehicle's flight mode is $Hover_Z$, it fails to maintain a constant altitude.	DISABLE	✓
Bug_{12}	Bebop2	$PP.HOME_1$	If $Mode_t$ is HOME and $Land_t$ is not true, then ALT_t is not ALT_{t-1} and Pos_t is not Pos_{t-1}	When nav_desend_v configuration parameter is 0, the vehicle fails to land on the ground.	CHECK	✓

TABLE VIII: Summary of new logic bugs discovered by PGFuzz on Paparazzi.

RVs through a remote controller channel. Yet, to trigger buggy behaviors, PGPATCH needs to fully manipulate inputs.

E. The Formulas' Usability

We evaluate PGPATCH on Paparazzi to find new logic bugs. To run PGFuzz on Paparazzi, we reused the five formulas used in PGFuzz for Paparazzi, and added a new formula to detect possible logic bugs. The new formula is for a fail-safe mode in Paparazzi: "If GPS_{loss} is true and RC_{valid} is false, then mode is FAILSAFE". The reason is that PGFuzz defines the fail-safe behavior of ArduPilot and PX4 in the form of LTL formulas but does not create a formula for Paparazzi's fail-safe mode.

We discovered a total of 12 previously unknown logic bugs by running PGFuzz on Paparazzi for a day (as shown in Table VIII). To detail, the PP.FailSafe, PP.Hover, and $PP.TAKEOFF_1$ formulas discovered multiple logic bugs (10 out of the 12 bugs). The $PP.Hover_Z$ and $PP.HOME_1$ each detected one bug (2 out of the 12 bugs). We have reported the discovered, previously unknown 12 logic bugs to Paparazzi developers, who acknowledged them.

By using the same formulas, PGPATCH fixed 6 out of the 12 logic bugs. Specifically, PGPATCH fixed 4 bugs of ADD type, 1 of DISABLE type, and 1 of CHECK type. PGPATCH failed to fix the other 6 logic bugs. The reason is that developers do not implement (*i*) Hover flight mode (Bug_5-Bug_8), and (*ii*) Quad_Elie0 and Quad-Navstik vehicle types (Bug_9 and Bug_{10}) although the documentation explicitly mentions that Paparazzi supports these features. These require PGPATCH to create the features from scratch.

F. Recruitment Details and Participant Demographics

We recruit from two different groups, RV developers and experienced RV users. The RV developers actively fix bugs reported in their GitHub repositories [8], [59], [64], community websites [9], [66], and live chat channels [10], [60], [67]. To recruit developers, we advertised our study in developer community websites and live chat channels. Experienced RV users also create patches, which are then reviewed and approved by developers before they are applied to the RV software. To recruit experienced RV users, we advertised our study through internal campus email listings and Slack channels to reach out to engineering and CS students familiar with RV software.

(a) Age				
Min = 19	Mean = 27.1	Median = 24.5	Stddev = 6	Max = 40
(b) Country of origin				
China = 7	S. Korea = 2	USA = 2		Pakistan = 1
(c) Achieved level of education				
Highschool = 2		Bachelor = 6		Graduate school = 4
(d) Do you know LTL syntax?				
Yes = 2				No = 10
(e) Which RV software have you modified?				
ArduPilot = 5		PX4 = 4		ArduPilot & PX4 = 3
(f) How many years of RV software experience do you have?				
Less than 1 year = 6		1 year = 3	2 years = 0	More than 3 years = 3
(g) What is your major?				
Computer science = 9		Electrical Engineering = 2		Aerospace Engineering = 1
(h) Do you primarily study/work in RVs?				
Yes = 4				No = 8
(i) What level of RV software programming proficiency do you think you have?				
Beginner = 7		Intermediate = 4		Advanced = 1

TABLE IX: Detailed data about demographics of experienced RV users (N = 12).

(a) Age				
Min = 22	Mean = 38.8	Median = 29.5	Stddev = 9.1	Max = 44
(b) Country of origin				
Brazil = 2	France = 1	Puerto Rico = 1	Turkey = 1	USA = 1
(c) Achieved level of education				
Highschool = 1		Bachelor = 2		Graduate school = 3
(d) Do you know LTL syntax?				
Yes = 0				No = 6
(e) Which RV software have you modified?				
ArduPilot = 6		PX4 = 0		ArduPilot & PX4 = 0
(f) How many years of RV software experience do you have?				
Less than 1 year = 0		1 year = 2	2 years = 1	More than 3 years = 3
(g) What is your major?				
Computer science = 2		Mechanical Engineering = 2	Aerospace Engineering = 1	N/A = 1
(h) Do you primarily study/work in RVs?				
Yes = 6				No = 0
(i) What level of RV software programming proficiency do you think you have?				
Beginner = 1		Intermediate = 3		Advanced = 2

TABLE X: Detailed data about demographics of experienced RV developers (N = 6).

ID	Formula	Description	# of fixed bugs
PP.FailSafe	If $\texttt{GPS}_{\texttt{loss}}$ is true and $\texttt{RC}_{\texttt{valid}}$ is false, then mode is FAILSAFE	FAILSAFE mode is triggered when the vehicle loses GPS and RC signals.	4
$\texttt{PP.Hover}_{\texttt{Z}}$	If $\texttt{Mode}_{\texttt{t}}$ is $\texttt{Hover}_{\texttt{Z}}$ and $\texttt{Throttle}_{\texttt{t}}$ is 1500, then $\texttt{ALT}_{\texttt{t}}$ is $\texttt{ALT}_{\texttt{t}-1}$	The vehicle must maintain a constant altitude if its flight mode is $\texttt{Hover}_{\texttt{Z}}$ and the current throttle is 1500.	2
RV.Safety *(generic formula)*	If $\texttt{Param}_{\texttt{i\_value}}$ is less than $\texttt{Min}_{\texttt{i}}$ or $\texttt{Param}_{\texttt{i\_value}}$ is more than $\texttt{Max}_{\texttt{i}}$, then Safety is error	The vehicle returns a safety error if i-th configuration parameter has a value outside its valid range.	44
RV.Check *(generic formula)*	If $\texttt{Param}_{\texttt{i\_value}}$ is less than $\texttt{Min}_{\texttt{i}}$ or $\texttt{Param}_{\texttt{i\_value}}$ is more than $\texttt{Max}_{\texttt{i}}$, then $\texttt{Param}_{\texttt{i\_value}}$ is $\texttt{Param}_{\texttt{i\_default}}$	When i-th configuration parameter have a value outside its valid range, then a patch assigns a default value to the parameter.	135

TABLE XI: Four PPL formulas fixing multiple bugs.

Our user study was approved by our institution's IRB [74], and considered exempt. We asked participants to fill out a consent form and answer demographic questions before conducting the user study. We compensated all participants with a $40 Amazon gift card.

22 participants (6 developers and 16 users) applied for our study, and 18 (6 developers and 12 users) qualified based on their experience in finding bugs in RVs and modifying RV software. We present the demographic data of the participants in Table IX and Table X. To the best of our knowledge, this is the first APR work that recruits experienced developers in the RV industry and compares their performance to the APR tool's one. Mainly, 1 of the 6 RV developers is an official maintainer of ArduPilot. He regularly contributes to the RV software and is responsible for reviewing patches on ArduPilot. He has more than 800 commit records dating back to the initial phases of ArduPilot. The other five developers have at least one year of experience in the RV industry.

G. Relationship between Formulas and Bugs

We used a total of 123 formulas to attempt to fix the 297 bugs in Section VII-A. 4 out of the 123 formulas (PP.FailSafe, $\texttt{PP.Hover}_{\texttt{Z}}$, RV.Check, RV.Safety) fixed multiple bugs, as shown in Table XI. The other 119 formulas could fix one bug each. We note that building formulas is still worthwhile even when one formula can fix only one logic bug. The reason is that creating formulas is much faster and less error-prone than manually patching bugs, as explained in Section VII-G.

Regarding the formulas patching multiple bugs, the PP.FailSafe formula ensures that the fail-safe mode is triggered when the RV loses GPS and RC signals. This formula patches four bugs. In this case, multiple patches are placed in separate locations in the code, depending on the RV's flight mode (e.g., AUTO or MANUAL) that triggers the bug.

The $\texttt{PP.Hover}_{\texttt{Z}}$ formula ensures that the RV maintains a constant altitude during the Hover_Z flight mode. This formula fixes two bugs. In this case, two identical patch code snippets are required in different functions according to vehicle types (multi-copter and fixed-wing).

Finally, RV.Safety and RV.Check are *generic formulas*, which means that some of their terms assume multiple values from a list. For instance, the RV.Safety formula uses different $\texttt{Param}_{\texttt{i}}$ and corresponding $\texttt{Min}_{\texttt{i}}$/$\texttt{Max}_{\texttt{i}}$ values, from a list of parameters and corresponding minimum/maximum values. Users can obtain a valid range for each of the configuration parameters from the official documentation [7], [58], [65]. An example of an entry of this list is: $< ANGLE\_MAX, 1000, 8000 >$. In this example, the RV.Safety formula will be instantiated to "If (ANGLE_MAX is less than 1000) or (ANGLE_MAX is more than 8000), then (Safety is error)".

In general, patches generated from the RV.Safety formula verify whether parameters related to altitude/attitude of the RV have a valid value. Using this formula, PGPatch inserts a patch verifying whether altitude/attitude-related parameters are within proper ranges. In case they are not, the patch sends a "Safety error" message to the GCS. Concretely, these patches address a bug causing the RV software not to send error messages to the GCS when in certain flight stages (e.g., after takeoff).

Similarly, patches generated using the RV.Check formula check if a non-altitude/attitude-related parameter is within a valid range. In the case of violations, the inserted patches assign the violating parameter to a default value. The difference between these two behaviors is justified by the fact that the RV software is designed to deal with invalid altitude/attitude-related parameters (although, in this case, it is supposed to send a message to the GCS), while it assumes that all other parameter values are always in valid ranges.