PatchVerif: Discovering Faulty Patches in 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

Modern software is continuously patched to fix bugs and security vulnerabilities. Patching is particularly important in robotic vehicles (RVs), in which safety and security bugs can cause severe physical damages. However, existing automated methods struggle to identify faulty patches in RVs, due to their inability to systematically determine patch-introduced behavioral modifications, which affect how the RV interacts with the physical environment.

In this paper, we introduce PATCHVERIF, an automated patch analysis framework. PATCHVERIF's goal is to evaluate whether a given patch introduces bugs in the patched RV control software. To this aim, PATCHVERIF uses a combination of static and dynamic analysis to measure how the analyzed patch affects the physical state of an RV. Specifically, PATCHVERIF uses a dedicated input mutation algorithm to generate RV inputs that maximize the behavioral differences (in the physical space) between the original code and the patched one. Using the collected information about patch-introduced behavioral modifications, PATCHVERIF employs support vector machines (SVMs) to infer whether a patch is faulty or correct.

We evaluated PATCHVERIF on two popular RV control software (ArduPilot and PX4), and it successfully identified faulty patches with an average precision and recall of 97.9% and 92.1%, respectively. Moreover, PATCHVERIF discovered 115 previously unknown bugs, 103 of which have been acknowledged, and 51 of them have already been fixed.

1 Introduction

Patching has been proven essential to fix bugs and security vulnerabilities [17,40,75,76]. However, recent works have shown that patches written by developers can be faulty [10, 54]. These faulty patches unintentionally break the intended functionality of software while attempting to fix existing issues or add new features [1, 8, 11, 38, 74].

This issue is particularly severe in software controlling robotic vehicles (RVs) for two main reasons. First, faulty patches in RVs can cause severe physical damage to critical infrastructures, humans, or be maliciously exploited by an attacker to stealthily disrupt an RV's behavior [9, 30, 32, 68, 69]. Second, detecting such faulty patches (manually or automatically) in RVs is particularly challenging. In fact, any patch to control software has potential consequences not only in the "cyber space" (i.e., how variables are used and how data flows in the software), but also in the "physical space" the device controlled by such software operates in.

Software developers are typically used to reasoning about patch-introduced modifications in the cyber space. For instance, a developer may check all the code locations in which a variable, whose initial value is changed by a patch, is used. On the contrary, tracking the patch-introduced behavioral modifications in the physical space is much harder. In our preliminary analysis of RV software, we found many cases in which the consequences of a code patch in the physical space were not fully understood by the developers. For instance, we found a case in which ArduPilot developers implemented a patch that updates conditions to trigger an emergency stop for helicopter-type vehicles. However, this patch also caused multicopter-type vehicles not to stop the motors, although the user activated the emergency stop [19].

Regarding automated analysis, researchers have proposed techniques to specifically detect faulty patches [35, 37, 56, 63, 73]. However, these techniques mainly focus on memory corruption bugs (e.g., buffer overflow), while in RV software, faulty patches could also cause logic bugs that lead to deviations in the RVs' physical behavior from the developer's expectations. Additionally, most of these techniques assume that a complete set of test cases exists. Unfortunately, available test cases only cover a limited subset of the possible behaviors of popular RV software, such as ArduPilot and PX4 [7, 52], since they are manually generated. Finally, recent works focus on specifically detecting bugs affecting the physical space in RV software [30, 32]. However, these techniques require a predetermined notion of what an RV's faulty behavior is. For instance, PGFuzz [30] assumes the existence of a set of temporal logic formulas precisely describing rules (related to physical properties) a vehicle should never violate.

In summary, two fundamental reasons make the detection of faulty patches in RV software challenging. First, the "input space" of RV software is much larger than that of normal software, given the amount of data that a vehicle receives and processes at any given point in time. This data encompasses environmental data, read continuously from multiple sensors, as well as commands given by ground control stations. For this reason, test suites[1] for RVs only cover a very limited subset of physical circumstances the vehicle may operate in. For example, ArduPilot and PX4 simulators consider 198 different environmental factors (e.g., terrain, temperature, and sensor failure). However, their test suites mainly test sensor failure circumstances [7,52]. Second, patch verification methods must know which behaviors of an RV are correct and which behaviors are anomalous. Yet, in RVs, there is no clear definition of a "correct behavior". In fact, the definition of what is correct is tightly related to both the physical space a device operates in and the assigned mission, and both of these concepts vary widely in different RV usage scenarios.

Motivated by the potential severity of patch-introduced bugs in RVs and the lack of effective approaches to automatically detect them, in this paper, we develop PATCHVERIF, an automated patch verification framework for RV software.

Our study reveals that a surprisingly high number of patches for the ArduPilot and PX4 control software are faulty. Specifically, we found 114 faulty patches affecting ArduPilot and 231 faulty patches affecting PX4. These faulty patches can be categorized into three groups: (*i*) partially fixing a buggy behavior, (*ii*) fixing an incorrect behavior but breaking another correct behavior, and (*iii*) adding a new feature but introducing a bug in existing features.

To find faulty patches, PATCHVERIF takes, as input, a patch and RV software. PATCHVERIF then determines how the analyzed patch affects physical properties of the RV. To comprehensively measure the patch-introduced variations of these properties, PATCHVERIF automatically generates test cases eliciting both the code locations and the physical behaviors that a patch affects. PATCHVERIF infers patch-introduced behavioral modifications via static and dynamic analysis. It then leverages code coverage and physical properties as guidance to test the patch under different circumstances. Through this analysis, PATCHVERIF generates the test cases that showcase how a patch affects the software both in the "cyber space" and "physical space". Finally, PATCHVERIF uses a machine learning approach to determine if the variations of the considered physical properties are symptoms of a new bug that has been introduced by the analyzed patch. If this is true, PATCHVERIF classifies the patch as faulty.

Different from recent related approaches [30, 32], PATCHVERIF focuses on physical properties, instead of a fixed or manually-specified notion of what a "correct behavior" is. For this reason, PATCHVERIF's approach is generic, both in terms of the type of bugs it can identify and the software code bases it can analyze.

We evaluated PATCHVERIF on two popular RV control software (ArduPilot and PX4) and successfully identified faulty patches with an average of 97.9% precision, 92.1% recall, and 94.9% F1-score. Further, PATCHVERIF revealed 115 previously unknown bugs caused by faulty patches. Out of these 115 bugs, developers acknowledge 103 bugs, and 51 of them have been fixed. In summary, we make the following contributions:

- **Behavior-aware Patch Profiling.** PATCHVERIF first identifies, among all possible "inputs" of an RV (user commands, configuration parameters, and environmental factors), those that are related to the analyzed patch. To trigger buggy behaviors caused by faulty patches, it then mutates the identified inputs, trying to maximize the variation of the considered physical properties.
- **Physical Invariant-aware Patch Verification.** We identify five physical properties of RV software whose excessive deviations are a symptom of a patch-introduced bug. Starting from this observation, we leverage machine learning to optimally determine if an observed variation, triggered by PATCHVERIF, of these properties is caused by a faulty patch.
- **Evaluation with Real-world RV Software.** We apply PATCHVERIF to two popular RV software packages: ArduPilot and PX4. PATCHVERIF successfully found faulty patches with an average F1-score of 94.9%. Moreover, it discovered 115 previously unknown bugs caused by faulty patches, 103 of which have been acknowledged by developers, and 51 of them have already been fixed.

PATCHVERIF is publicly available at https://github.com/purseclab/PatchVerif.

2 Background

Faulty Patches. The faulty patches that we target can happen due to three main reasons; a patch (*i*) partially fixes a buggy behavior, (*ii*) fixes an incorrect behavior but hurting another correct behavior, and (*iii*) adds a new feature but introduces a bug. The faulty patches can cause two types of bugs: (1) memory bugs that violate memory access safety stopping program execution and (2) logic bugs that make the RV control software behave incorrectly. While we mainly target identifying faulty patches that cause logic bugs, PATCHVERIF naturally triggers and detects the memory bugs (e.g., null pointer dereferences) as well since it mutates inputs.

Input Space in RVs. To test whether a patched RV program behaves correctly in various scenarios, we must mutate the RV's behaviors and environmental conditions. The reason is that some buggy behaviors caused by faulty patches only appear in certain physical environmental situations (e.g., changing flight mode on drastic terrain variation, as explained in Section 8.6.2). Mutating three types of inputs results in

[1] Test suites, manually-defined by RV developers, consist of test cases to verify whether patched software correctly works under different scenarios.

Listing 1 A faulty patch [60] removes line 2 and adds line 3.

```
1   void Mode::navigate_to_waypoint() {
2 -   float desired_speed = g2.wp_nav.get_speed();
3 +   float desired_speed = g2.wp_nav.get_desired_speed();
```

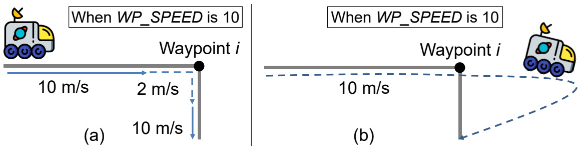


Figure 1: (a) Normal RV behavior during a pivot turn and (b) abnormal RV behavior after deploying a faulty patch.

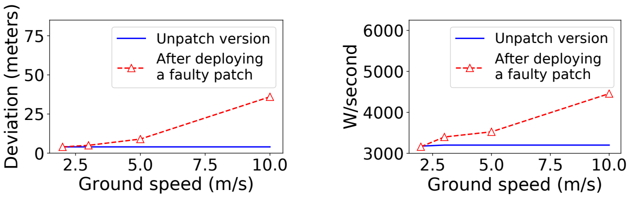
different scenarios. (1) User commands are used to directly operate RVs, e.g., turning left or right. (2) Configuration parameters configure the control algorithm in the RV software affecting behaviors and decision making, e.g., setting a threshold to stop the vehicle. (3) Environmental factors (e.g., wind and terrain) physically affect the vehicle's behavior in fields.

3 Motivating Example

We provide a faulty patch in ArduPilot that PATCHVERIF targets to discover. ArduPilot supports a pivot turn for rovers [46]. When a rover is near a waypoint (i.e., a corner), the vehicle reduces its speed or stops, turns towards the next waypoint, and continues the navigation. The rover can (*i*) decrease deviation from a planned path and (*ii*) prevent rollover accidents through the pivot turn, as shown in Figure 1-(a). However, a bug makes the rover increase its speed regardless of the pilot's input. To fix this bug, ArduPilot developers created a patch [60] to make the vehicle maintain a constant speed. Yet, the patch includes a buggy code because it enforces the vehicle to maintain a constant ground speed even near a corner, as shown in Figure 1-(b) and Listing 1. To detail, the `desired_speed` decides the vehicle's ground speed (Line 2), and `g2.wp_nav.get_speed()` returns a limited speed so that the vehicle slows down before reaching a corner. The faulty patch enforces the `desired_speed` to have a constant value (`g2.wp_nav.get_desired_speed()` returns the `WP_SPEED` configuration parameter value) (Line 3). As a result, the vehicle deviates from the planned path (i.e., increasing position errors) due to the high speed at the corner if the ground speed set by the `WP_SPEED` parameter is high, as shown in Figure 1-(b).

The developers noticed the buggy behavior after around three months of deploying the faulty patch [47]. This happens because ArduPilot's test suite tests the pivot turn with a default ground speed [7]. Yet, the default speed is insufficient to make the vehicle deviate from the planned path, as the vehicle driving at a low speed does not need to slow down at corners.

PATCHVERIF conducts the following steps to detect such a faulty patch. (1) Identifying the RV's states that are affected by the patch and the environmental conditions that impact the patch by analyzing the removed/added code snippets and developer comments in the patch code. For example, the patch code in Listing 1 changes the RV's *speed*, and outdoor wind affects the patch (i.e., *speed*). (2) Determining the inputs (i.e., configuration parameters, user commands, environmental factors) that affect the identified RV states and environmental conditions. For example, `WP_SPEED` configuration parameter and `WIND_SPEED` environmental factor directly change the RV's *speed*. (3) Finding inputs triggering the patch code snippet. For instance, to trigger the patch code in Listing 1, PATCHVERIF adds a waypoint and changes the vehicle's flight mode to `AUTO`. (4) Mutating the identified inputs to generate test cases which are highly likely to make the patched software deviate from normal behaviors if the patch is faulty. For instance, PATCHVERIF assigns a larger value to `WP_SPEED` to make the vehicle deviate from the planned path due to the increasing *speed* at the corner, as shown in Figure 2. (5) Concluding that a patch is faulty if the patched program violates one of the five physical invariant properties in RV control programs. For example, the faulty patch in Listing 1 makes the patched RV control program violate a physical invariant property that states: "the patch must not increase position errors", as shown in Figure 2a.



(a) Deviation from a planned path.

(b) Battery consumption.

Figure 2: Deviated RV behaviors after deploying the faulty patch.

4 Threat Model

We consider faulty patches, containing logic bugs, created by developers or adversaries. Developers may unintentionally produce faulty patches for various reasons, such as unexpected corner cases (e.g., drastic terrain variation and strong wind), misunderstanding of design, and insufficient documentation. Adversaries may also create and inject seemingly innocent patches, which are actually faulty, into open-source software. Such faulty patch injection attacks have already been investigated in popular software, such as the Linux kernel [11, 38].

Logic bugs in RV control software have recently received significant attention from security researchers due to the large attack surface in RVs and significant physical damage caused by the logic bugs [12, 30–32]. Indeed, our threat model is in line with prior works on RV security such as PGFuzz [30], RVFuzzer [32], CPI [12], and PGPatch [31].

While, in theory, an attacker could simply spoof self-destructive commands (e.g., stopping actuators or assigning extreme values to inputs), in practice, these attacks are easy to notice through either run-time or offline flight logs [6, 50]. In contrast, exploiting buggy behaviors introduced by faulty patches enables conducting more stealthy attacks that do not necessarily require sending spoofed commands (as we will

Physical invariants	Prior works on RVs	Test suites created by developers [7, 52]	Our observation on patches
Timeliness	✓ [28]	✓	✓
Precise navigation	✓ [13, 30, 32, 55]	✓	✓
Stability	✓ [13, 30, 32, 55]	✓	✓
State consistency	✗	✓	✓
Efficiency	✗	✗	✓

Table 1: Origin of the five physical invariants leveraged by PatchVerif.

evaluate later in Section 9).

5 Design Challenges

Developing an automatic patch verification framework, which handles prior works' limitations, raises unique challenges.

C1: Generality. The first challenge concerns determining physical invariant properties that can always be applied to all patches. To detect faulty patches without users' manual effort, PatchVerif leverages five physical invariants as a bug oracle. PatchVerif expects that a correct patch must not (1) increase mission completion time (`Timeliness`), (2) increase battery consumption (`Efficiency`), (3) increase position errors (`Precise navigation`), (4) increase instability (`Stability`), and (5) change unexpected RV states (`State consistency`).

These invariants are inspired by (*i*) prior works on RVs [13, 28, 30, 32, 55], (*ii*) test suites created by RV developers [7, 52], and (*iii*) our observation of correct/faulty patches in RV control software, as shown in Table 1.

Particularly, many prior works on RVs leverage `Timeliness`, `Precise navigation`, and/or `Stability` to detect either buggy behaviors or physical attacks (e.g., GPS spoofing and acoustic sensor attacks) [13, 28, 30, 32, 55]. For instance, PGFuzz and RVFuzzer [30, 32] use `Precise navigation` and `Stability` to detect logic bugs in RV control software because buggy behaviors caused by the logic bugs increase position errors and/or instability.

Test suites [7, 52] created by RV developers leverage `Timeliness`, `Precise navigation`, `Stability`, and `State consistency` to decide whether the patched RV software passes the test cases. We further include battery consumption as a new physical invariant (`Efficiency`) for the following reasons. (1) Many patches affect battery efficiency. (2) It helps PatchVerif more clearly distinguish between correct and faulty patches. In Appendix A, we present experimental results to justify the selection of these invariants.

However, during our analysis of patches, we notice that some correct patches naturally break the physical invariant properties. For example, a patch makes an RV wait for a specific time to proceed with the following mission item [15]. In such a case, executing the delay command increases the time required to finish a mission and battery consumption. However, the patch does not include any buggy code. To address this issue, PatchVerif first identifies the patch type; it then selectively applies five properties to detect bugs (Detailed in Section 6.5.2).

Listing 2 A patch implementing *terrain-following* for the `CIRCLE` flight mode. This patch contains buggy code.

```
+void AC_Circle::set_center(const Location& center) {
+  if (center.get_alt_frame() == ABOVE_TERRAIN) {
+    if (center.get_vector_xy_from_origin(center_xy)) { ... }
+    else { ... } }
+  else {
+    if (!center.get_vector_from_origin(circle_center)) { ... }
```

C2: Natural Deviations. The second challenge is that the RVs' operating environment may naturally cause deviations from a planned mission similar to the symptoms of a faulty patch. An RV shows slightly different mission completion time, battery consumption, and instability due to environmental changes (e.g., wind), though it uses the same software version. To address this, PatchVerif leverages a supervised classification algorithm to distinguish normal behaviors and symptoms of faulty patches (Detailed in Section 6.5.2).

C3: Patch Coverage. The third challenge concerns how to verify all patch-introduced behavioral modifications. To address this issue, PatchVerif uses a white-box fuzzing method to extract inputs that execute the patch code. A test case created by PatchVerif always includes the extracted inputs. Further, PatchVerif infers which physical states of the RV are affected by the patch and which environmental conditions affect the patch. It then includes inputs, which change the RV's physical states and environmental conditions that might affect the patch, to the test case. Yet, a patch may contain an unreachable conditional branch triggered by a specific environment and context-independent inputs, e.g., loading damaged terrain data to an RV. To execute and test the unreachable branch, PatchVerif leverages symbolic execution (Detailed in Section 6.3).

C4: Triggering Bugs. The last challenge is how to generate test cases that are highly likely to trigger buggy codes created by the faulty patches. Randomly executing inputs, which are related to a patch, is unlikely to trigger the bugs. To handle this issue, PatchVerif mutates inputs based on code coverage and distance metrics to make the patched software violate one of the five physical invariant properties if the patch includes buggy code (Detailed in Section 6.5.3).

6 PatchVerif

Figure 3 shows the architecture of PatchVerif, which is mainly composed of three parts. PatchVerif takes a patch code snippet in C/C++ as input (❶), and then (1) identifies inputs related to the patch, (2) analyzes patch type, and (3) mutates the identified inputs to create a test case that triggers a buggy behavior if the patch contains a flaw.

6.1 Analyzing Physical Impact of a Patch

PatchVerif analyzes an RV's states that are affected by the patch and environmental conditions that affect the patch (❷ in Figure 3). We use the RV's states and environmental con-

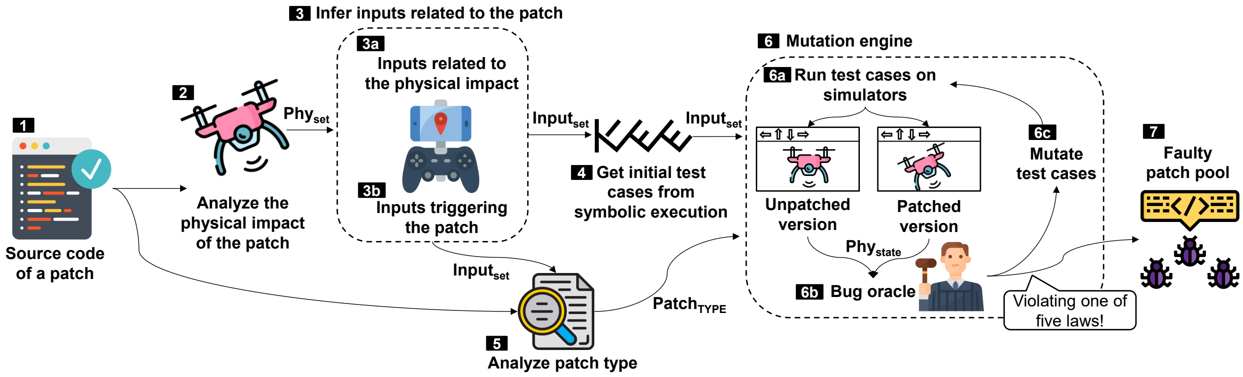
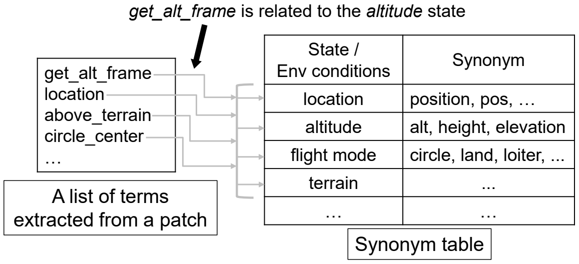

1
Source code of a patch
2
Analyze the physical impact of the patch
Phy_{set}
3 Infer inputs related to the patch
3a
Inputs related to the physical impact
3b
Inputs triggering the patch
$\text{Input}_{\text{set}}$
4 Get initial test cases from symbolic execution
$\text{Input}_{\text{set}}$
$\text{Input}_{\text{set}}$
5
Analyze patch type
$\text{Patch}_{\text{TYPE}}$
6 Mutation engine
6a Run test cases on simulators
Unpatched version
Patched version
$\text{Phy}_{\text{state}}$
6b Bug oracle
6c
Mutate test cases
Violating one of five laws!
7
Faulty patch pool

Figure 3: Overview of PatchVerif's workflow.


get_alt_frame is related to the *altitude* state
get_alt_frame
location
above_terrain
circle_center
…
A list of terms extracted from a patch
Synonym table

State / Env conditions	Synonym
location	position, pos, …
altitude	alt, height, elevation
flight mode	circle, land, loiter, ...
terrain	…
…	…

Figure 4: Illustration of the name matching-based static analysis.

ditions for choosing inputs used to construct test cases. We leverage a name-matching method because variable/function names in RV software are decided by strict coding conventions [2]. Unlike general software, the names of variables/-functions must match with the RV's physical states and/or environmental conditions. Particularly, PatchVerif conducts the following steps: (1) extracts the names of variables and functions in the patch, which includes removed/added code snippets and developer comments, (2) filters out all but nouns from the variable/function names and comments, and (3) matches the extracted terms with RV physical states and environmental conditions in the synonym table (Figure 4).

We adapt a list of the RV's states and environmental conditions from PGFuzz [30]. For example, when PatchVerif analyzes the physical impact of the patch in Listing 2, PatchVerif first extracts `location`, `get_alt_frame`, `ABOVE_TERRAIN`, and `circle_center`, etc. It then matches these terms to the list of the RV's states and environmental conditions. PatchVerif concludes that the patch in Listing 2 changes the RV's *location*, *altitude*, and *CIRCLE flight mode* states, and *terrain* as an environmental condition that affects the patch. We refer to the matched RV states and environmental conditions as $\texttt{Phy}_{\texttt{set}}$.

The name-matching method fails to match a patch with correct physical states if an RV software does not follow the coding conventions [2]. To tackle this, PatchVerif allows users to use a taint analysis instead of the name-matching method. The taint analysis tracks all data propagation from the patch code snippet (Detailed in Appendix C). Yet, PatchVerif integrates the name-matching method as the default option because (*i*) our target RV software strictly follows the coding conventions, and (*ii*) the name-matching method shows better performance than the taint analysis (Section 9).

6.2 Inferring Inputs Related to the Patch

PatchVerif infers specific inputs that (*i*) change an RV's states impacted by the patch and environmental conditions that affect the patch, and (*ii*) execute a patch code snippet.

Inputs Related to the Physical Impact. PatchVerif obtains a list of the RV's states and environmental conditions related to a patch via static analysis (Section 6.1), e.g., the *location*, *altitude*, and *CIRCLE flight mode* as the RV's states and *terrain* as the environmental condition. We extend PGFuzz's profiling engine [30] to support all RV states. It allows us to map inputs to the RV's states and environmental conditions. PatchVerif stores the mapped inputs into $\texttt{Input}_{\texttt{set}}$.

Inputs Triggering the Patch. PatchVerif identifies a set of inputs that trigger a patch code snippet, and its mutation engine later uses the identified inputs to create test cases. To find the inputs triggering the patch code, PatchVerif conducts the following steps: (1) It first builds RV control software with the code coverage analysis tools of Gcov [22] and LCOV [36]. (2) It then randomly selects and executes an input on the simulator to find whether an input triggers a function that contains the patch. For instance, PatchVerif discovers that executing `MAV_CMD_DO_SET_MODE` user command (changing the flight mode) allows us to execute `AC_Circle::set_center()` in the faulty patch in Listing 2. (3) Lastly, PatchVerif stores the input into $\texttt{Input}_{\texttt{set}}$ if the input executes the patch code. We will detail the performance of finding patch-triggering inputs in Section 9.

6.3 Creating Test Cases

PatchVerif targets two goals to address patch coverage (*C3* in Section 5): (*i*) identifying inputs triggering conditional branches (i.e., "if statements") in the patch, and (*ii*) generating initial test cases based on the identified inputs (❸ in Figure 3). PatchVerif obtains an input, which triggers a function containing the patch, e.g., from Section 6.2, changing the flight mode allows PatchVerif to execute a function with the faulty patch in Listing 2. Yet, to test all behaviors affected by the patch, PatchVerif targets to test all conditional branches in the patch code.

Triggering Reachable Branches. To find inputs triggering the "if statements", PatchVerif randomly selects an input that changes one of $\texttt{Phy}_{\texttt{set}}$ and executes the selected input on a simulator. If an input triggers a conditional branch in the patch, PatchVerif stores the input to $\texttt{Input}_{\texttt{set}}$. For instance, PatchVerif triggers the conditional branch at line 2 in Listing 2 by executing `MAV_CMD_NAV_LOITER_TURNS` user command with the terrain altitude frame option.

Triggering Unreachable Branches using Symbolic Execution. PatchVerif might find conditional branches in the patch that cannot be triggered even after executing inputs for a user-defined time. We refer to those "if statements" as $\texttt{if}_{\texttt{un}}$.

To address this issue, PatchVerif (1) extracts terms (i.e., variables/function calls) from the guarding conditions of every $\texttt{if}_{\texttt{un}}$, (2) finds the definition site for each term, and (3) leverages symbolic execution (❹ in Figure 3) to determine assignments of these terms that allow reaching the unreachable location. In particular, these steps are automated by replacing the terms with symbolic variables using KLEE's `klee_make_symbolic` API calls.

We note that PatchVerif runs the symbolic execution only if the patch includes an $\texttt{if}_{\texttt{un}}$, and it does not run symbolic execution on whole software. Instead, PatchVerif first extracts the function that includes the patch code snippet. It then runs the symbolic execution engine KLEE [33] to determine concrete variable assignments (for local variables, global variables, and called functions) to trigger $\texttt{if}_{\texttt{un}}$.

To insert the found concrete assignments into the patch, PatchVerif assigns constants obtained from the symbolic execution's test cases into local variables, e.g., $\texttt{var} = 20$; $\texttt{if}_{\texttt{un}}(\texttt{var} > 10)$, and it replaces function calls and global variables with constants, e.g., replacing $\texttt{if}_{\texttt{un}}(\texttt{func}())$ with $\texttt{if}_{\texttt{un}}(1)$. These constants are obtained from the test cases created by the symbolic execution.

We note that the triggered unreachable branch may prevent triggering the other conditional branches in the patch code snippet. In the following patch code: $\texttt{if}_{\texttt{un}}(1)\{\texttt{return true};\}\ \texttt{if}(\texttt{C})\{...\}$, the concrete assignment (i.e., 1) triggers the unreachable branch but prevents PatchVerif from testing another branch (i.e., $\texttt{if}(\texttt{C})\{...\}$) because the triggered branch terminates its function. To test the rest of the code lines in the patch, PatchVerif tests the correctness of the original patch created by developers and the patch modified by PatchVerif. Triggering unreachable branches enables us to increase code coverage and discover six additional bugs introduced by faulty patches in ArduPilot and PX4.

We restrict the exploration time to a maximum of 30 minutes (i.e., the `-max-time` KLEE option is 30), and we set the `-max-depth` KLEE option to 5 to limit the maximum number of branches explored in unbounded paths.

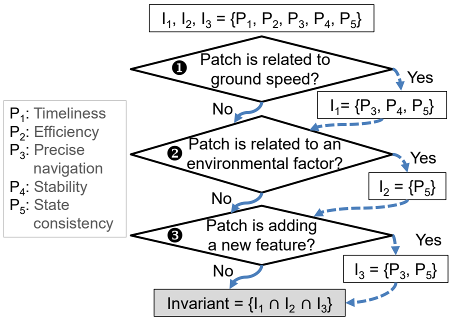

I1, I2, I3 = {P1, P2, P3, P4, P5}
❶ Patch is related to ground speed?
Yes
I1= {P3, P4, P5}
No
❷ Patch is related to an environmental factor?
Yes
I2 = {P5}
No
❸ Patch is adding a new feature?
Yes
I3 = {P3, P5}
No
Invariant = {I1 ∩ I2 ∩ I3}
P1: Timeliness
P2: Efficiency
P3: Precise navigation
P4: Stability
P5: State consistency

Figure 5: Criteria to selectively apply the five physical invariants.

6.4 Analyzing Patch Type

PatchVerif classifies the patch into one of four types (❺ in Figure 3): (1) changing the RV's ground speed ($\texttt{Patch}_{\texttt{speed}}$), (2) related to an environmental factor, e.g., wind ($\texttt{Patch}_{\texttt{env}}$), (3) adding a new feature ($\texttt{Patch}_{\texttt{new}}$), and (4) other patch types ($\texttt{Patch}_{\texttt{other}}$). PatchVerif's bug oracle leverages the patch type to selectively apply the five properties for detecting bugs' symptoms. We note that a patch can be classified into multiple patch types.

To determine whether the patch changes the RV's *speed*, PatchVerif leverages $\texttt{Phy}_{\texttt{set}}$ from the identified physical impact of the patch (Section 6.1). For instance, it discovers that the RV's *speed* state is related to the faulty patch in Listing 1 through static analysis. To decide whether an environmental factor is related to a patch, we manually construct a list of environmental factors from RV documentation [4, 48]. PatchVerif concludes that the patch affects an environmental factor if the patch code includes one of the terms in the list of environmental factors. Third, PatchVerif obtains an input to trigger the patch code from $\texttt{Input}_{\texttt{set}}$. PatchVerif then executes the obtained input on an unpatched and patched version on a simulator. We assume the patch implements a new feature if two conditions are satisfied: (1) there is no difference in the RV's states before and after executing those inputs on the unpatched version, and (2) the RV's physical states are changed after executing the inputs on the patched version. Lastly, PatchVerif concludes that a patch is included in $\texttt{Patch}_{\texttt{other}}$ if the patch is not included in the above three categories.

6.5 Mutation Engine

Mutation Engine Overview. Algorithm 1 details the steps of PatchVerif's mutation engine (❻ in Figure 3). The algorithm repeatedly conducts the following. (1) It obtains inputs that trigger a patch code snippet (Line 14). (2) It randomly selects inputs from $\texttt{Input}_{\texttt{set}}$ (Line 15). (3) It picks values based on the guidance (Line 16), e.g., assigning a high value to `WP_SPEED` enlarges differences in physical invariant properties between the unpatched and the patched versions in Figure 1. Thus, it keeps increasing `WP_SPEED`'s value. (4) It returns a test case consisting of the selected inputs and values (Lines 17 and 18). (5) It next initializes the simulator

Algorithm 1 PATCHVERIF's mutation engine

```
Input: A patch $P$, physical invariants $PI$, a patched RV $RV_{new}$, an unpatched RV $RV_{old}$,
a time limit $\tau$, a simulator $SIM$, RV's states related to the patch $Phy_{set}$, inputs related to
the patch $Input_{set}$, patch type $P_{type}$
Output: Faulty patch $FP$, a test case triggering buggy behaviors $Test_{case}$
 1: function PATCHWARNING($P$, $\tau$)                                        ▷ Main
 2:     while $C > 0$ or $time < \tau$ or $FP = \emptyset$ do
 3:         $Test_{case} \leftarrow$ MUTATE($Input_{set}$, $P$, $DIS$)                ▷ Get mutated inputs
 4:         $SIM_{<1,2>} \leftarrow SIM$.initialize()                         ▷ Initialize simulators
 5:         $SIM_{<1,2>} \leftarrow SIM$.env($Phy_{set}$)                      ▷ Change environment
 6:         $PI_{old} \leftarrow SIM_1$.execute($Test_{case}$, $RV_{old}$)             ▷ Collect RV's states
 7:         $PI_{new} \leftarrow SIM_2$.execute($Test_{case}$, $RV_{new}$)             ▷ Collect RV's states
 8:         $DIS$, $FP \leftarrow$ ORACLE($PI_{old}$, $PI_{new}$, $P_{type}$)            ▷ Calculate distances
 9:         $C \leftarrow$ UPDATE_COVERAGE($RV_{old}$, $RV_{new}$)                 ▷ Update coverage
10:     end while
11:     return $\langle FP, Test_{case} \rangle$
12: end function
13: function MUTATE($P$, $Input_{set}$, $DIS$, $C$)                          ▷ Mutating inputs
14:     $input_m \leftarrow$ MANDATE($Input_{set}$, $P$)                    ▷ Inputs triggering P's code
15:     $I \leftarrow$ RANDOM($Input_{set}$)                   ▷ Pick random inputs from $Input_{set}$
16:     $I \leftarrow$ GUIDANCE($I$, $DIS$, $C$)                       ▷ Pick values based on $DIS$ and $C$
17:     $Test_{case} = input_m \cup I$
18:     return $Test_{case}$
19: end function
```

and randomly changes environmental conditions (e.g., terrain) from $\mathtt{Phy_{set}}$ (Lines 4 and 5). (6) It executes the test case on both unpatched and patched versions on separate simulators, and records physical invariant properties (e.g., battery consumption) from both simulators (Lines 6 and 7). (7) It gets differences between the unpatched version's physical invariant properties and patched versions' ones. PATCHVERIF's bug oracle then decides whether such differences are symptoms of a faulty patch (Line 8). (8) It obtains updated code coverage according to the executed test case (Line 9), which is used to terminate the algorithm. The algorithm is ended when one of the following two cases occurs: (*i*) the bug oracle classifies the patch as a faulty patch, and (*ii*) it cannot detect any progress in code coverage for a user-defined time.

6.5.1 Running Test Cases

To compare the behavioral differences between unpatched and patched code versions, we choose the released version closest to the patch as the unpatched RV software. The reason is that RV developers disclose the released version only after numerous iterations of flight tests in fields. If a patch's type is $\mathtt{Patch_{new}}$, we compare the patched version with the version just before applying the patch.

6.5.2 Bug Oracle

Selecting Invariants. PATCHVERIF determines the invariants for detecting faulty patches by traversing the flow diagram in Figure 5 based on the inferred patch type in Section 6.4. This addresses the design challenge *C1: Generality* (Section 5) since all five invariants do not apply to all patch types. First, if a patch is related to *speed*, we exclude the "mission time" ($\mathtt{P_1}$) and "battery consumption" ($\mathtt{P_2}$) because the *speed* changed by the patch naturally changes $\mathtt{P_1}$ and $\mathtt{P_2}$ regardless of the patch's correctness. Second, if the patch affects an environmental factor, PATCHVERIF only leverages the "expected vehicle state change" ($\mathtt{P_5}$) to detect a bug's symptoms because the environment changed by the patch affects all properties ($\mathtt{P_1}$-$\mathtt{P_4}$). However, the environmental factors do not change unexpected vehicle states (e.g., sensor failure or crashing on the ground) if the patch is correct. Lastly, we exclude $\mathtt{P_1}$, $\mathtt{P_2}$, and "stability" ($\mathtt{P_4}$) if the patch adds a new feature into the RV software. The reason is that the newly supported feature could directly change these invariants.

Feature Engineering. We consider the RV's states related to *sensors*, *flight mode*, *pre-arm check*, and *flight stage* as features of `State consistency` ($\mathtt{P_5}$) invariant property. This is because the deviation in such RV's states leads to dangerous flight behaviors, e.g., sensor failure or crashing on the ground. We leverage the following features as the four physical invariant properties: (1) subtracting a mission start time from mission end time as a feature of `Timeliness` ($\mathtt{P_1}$), (2) current drawn from the battery in $\mathtt{amperage} \times \mathtt{100}$ as a feature of `Efficiency` ($\mathtt{P_2}$), (3) subtracting actual position (in Cartesian coordinate system) from target position as a feature of `Precise navigation` ($\mathtt{P_3}$), (4) adding instability, which is calculated by subtracting actual attitude (i.e., roll, pitch, and yaw angles) from target attitude, and vibration as a feature of `Stability` ($\mathtt{P_4}$). These features are detailed in Section 7.

Bug oracle. To handle *C2: Natural Deviations* (Section 5), we leverage a supervised classification algorithm that takes four features ($\mathtt{P_1}$-$\mathtt{P_4}$) labeled with "correct" and "faulty" classes. We construct the dataset with the features extracted from known faulty patches, and correct patches. After we learn a model from the dataset, we use the model to evaluate features of unknown patches to detect faulty patches.

6.5.3 Mutating Test Cases

Test Case Template. A test case is not feasible if it consists of a purely random order of inputs (e.g., increasing altitude before take-off). To resolve this, we use a mission plan obtained from ArduPilot's test suites [7] as a template to mutate test cases. PATCHVERIF mutates inputs on top of the mission.

Test Case Mutation. To address *C4: Triggering Bugs*, PATCHVERIF leverages the following distance metric: $\mathtt{DIS} = \sum_{i=1}^{4} |\mathtt{F_i} - \mathtt{F'_i}|$, where $\mathtt{F_i}$ and $\mathtt{F'_i}$ represent physical invariants in patched and unpatched versions, respectively. PATCHVERIF aims to maximize the distance, i.e., $\mathtt{max}(\sum_{i=1}^{4} |\mathtt{F_i} - \mathtt{F'_i}|)$. To achieve this, PATCHVERIF intelligently assigns values to inputs and executes them on a simulator. PATCHVERIF's mutation method guides the input generation based on whether executing an input causes increasing or decreasing the distance (`DIS`) values. For instance, if PATCHVERIF assigns a value higher than the default value to an input, and the executed input increases `DIS`, PATCHVERIF keeps increasing this input's value. Further, if an input, which has a value outside the valid range, makes the RV fail to complete a test case on both the patched and the unpatched versions, PATCHVERIF considers such an input assignment as self-sabotaging and enforces the input to have a value within the valid range. To obtain valid ranges of inputs, we adopt the PGFuzz's Pre-Processing component [30].

7 Implementation

Analyzing a Patch. To analyze (*i*) which RV's states are affected by patches and (*ii*) which environmental conditions might affect the patches, we write 379 lines of code (LoC) in Python. We adopt a list of the RV's states and environmental conditions from PGFuzz [30]. We modify/add 132 LoC from PGFuzz's profiling engine to obtain inputs that change the inferred RV's states and environmental conditions. We build ArduPilot and PX4 with Gcov v7.5 [22] and LCOV v1.13 [36] to support code coverage. To infer inputs that trigger patch code, we modify/add 681 LoC in test suites of ArduPilot v4.1 and PX4 v1.13. To make patch code snippets run with KLEE v2.2 [33], we write 406 LoC in Python.

Analyzing Patch Type. We write 853 LoC using the Pymavlink v2.4.16 library [53] that makes PATCHVERIF communicate with simulated RVs through MAVLink [42]. To run the patch type analyzer for PX4, we additionally modify/add 109 LoC of the patch type analyzer for ArduPilot because ArduPilot and PX4 differently implement MAVLink protocol. To simulate RVs, we select Software in the Loop (SITL) [59] for ArduPilot and Gazebo v11 [21] for PX4. Further, we manually construct a list of environmental factors from documentation of ArduPilot and PX4 [4, 48].

Feature Engineering. ArduPilot and PX4 store flight data to files (i.e., vehicle's black box) [6, 50]. PATCHVERIF parses the flight data to obtain features of the five invariant properties: `Timeliness`, `Efficiency`, `Precise navigation`, `Stability`, and `State consistency` as shown in Table 2. Particularly, PX4 does not calculate estimated battery consumption on a simulator, unlike ArduPilot. To infer the battery consumption on PX4, we leverage pulse width modulation (PWM) signals sent to actuators because the battery consumption is mainly proportional to the strength of the PWM.

Mutation Engine. We leverage support-vector machines (SVMs) [14] as PATCHVERIF's bug oracle instead of other classification algorithms such as decision trees [62] and deep neural networks (DNNs) [23] for the following reasons. (1) Decision trees are well known to show the best inference accuracy for categorical data. However, our four physical invariants are numerical (continuous) variables, and one is a discrete variable. (2) DNNs can achieve high performance, but they require a large dataset (e.g., tens of thousands) for training. Yet, we could not find such a large set of faulty patches from ArduPilot and PX4.

To evaluate these hypotheses, we performed experiments (Detailed in Appendix B.1 and Appendix B.2) in which we used decision trees and DNNs in place of SVMs. SVMs yield a 94.9% F-1 score, and the F1-score of decision trees and DNNs was 93.1% and 89.4%, respectively.

To build our bug oracle, we train a separate SVM classifier per RV control software because we extract features from different RV states of each RV program. We write 397 LoC in Python for training and testing patches through SVM classifiers. We note that although we leverage SVMs, PATCHVERIF allows users to easily change the bug oracle's classifier (Detailed in Appendix B.3). To mutate and run test cases, we write 1,525 and 1,630 LoC in Python on Pymavlink v2.4.16. for ArduPilot and PX4, respectively.

8 Evaluation

We evaluated PATCHVERIF on the two most popular RV control software: ArduPilot and PX4.

8.1 Evaluation Setup

To create a ground truth of correct and faulty patches, we used the following methods.

Collecting Faulty Patches. Reading commit messages from the Git history of both ArduPilot and PX4, we found that developers often refer to specific patches, mentioning they are faulty. Thus, we selected all patches the developers mention as faulty in the Git history. This process gave us an initial set ($\texttt{Patch}_{\texttt{F}}$) of 345 patches. Among these faulty patches, we focused on the ones satisfying the following criteria:

1. The bug that they introduce can be reproduced on a simulator. To reproduce bugs, we exploited commit messages from the Git history.
2. They have been reported within the last three years. We exclude patches that are too old, since their related bugs become increasingly harder to reproduce in a simulator, due to modifications in the simulator.

Out of all the patches satisfying these two criteria, we created a set ($\texttt{TrainSet}_{\texttt{F}}$) of 80 faulty patches (40 for ArduPilot and 40 for PX4).

Selecting Correct Patches. From the Git repositories of ArduPilot and PX4, we created a set of correct patches, selecting patches satisfying the following criteria:

1. The patched code can be triggered on a simulator.
2. The patch is older than one year. In fact, we realized that very recent patches are often "temporary" and incomplete. For this reason, their functionality is likely to be modified by another subsequent patch.
3. The patch was analyzed by two of the authors of this paper, who agreed on its correctness.

Among the patches that satisfied these criteria, we randomly selected a set of 80 correct patches ($\texttt{TrainSet}_{\texttt{C}}$), containing 40 patches for ArduPilot and 40 for PX4.

Patches for Evaluating PATCHVERIF's Bug-discovery Capabilities. We separately created another dataset containing patches for which we did not know whether they are faulty or correct. We used this dataset to evaluate PATCHVERIF's ability to discover new, previously unknown bugs caused by faulty patches. To create this dataset, we randomly selected 1,000 patches (500 for ArduPilot and 500 for PX4), satisfying the following criteria:

1. The patch does not belong to the $\texttt{TrainSet}_{\texttt{F}}$ or $\texttt{TrainSet}_{\texttt{C}}$ set.
2. The patched code can be triggered on a simulator.

<table>
<tr><th>Invariant property</th><th>Description</th><th>ArduPilot</th><th>PX4</th></tr>
<tr><td><code>Timeliness</code></td><td>Subtracting a mission start time from mission end time</td><td colspan="2">Landing time - Take off time</td></tr>
<tr><td><code>Efficiency</code></td><td>Current drawn from the battery</td><td>Curr</td><td>Output 0 - 6</td></tr>
<tr><td><code>Precise navigation</code></td><td>Subtracting actual position from target position</td><td>|TPX - PX|
+ |TPY - PY|
+ |TPZ - PZ|</td><td>|X setpoint - X estimated|
+ |Y setpoint - Y estimated|
+ |Z setpoint - Z estimated|</td></tr>
<tr><td><code>Stability</code></td><td>Subtracting actual attitude from target attitude</td><td>|DesRoll - Roll|
+ |DesPitch - Pitch|
+ |DesYaw - Yaw|</td><td>|Roll rate integral - Roll rate estimated|
+ |Pitch rate integral - Pitch rate estimated|
+ |Yaw rate integral - Yaw rate estimated|</td></tr>
<tr><td></td><td>Vibration calculated from standard deviation of the primary accelerometer's output in m/s/s</td><td>Vibe X + Vibe Y + Vibe Z</td><td>Vibration Metrics</td></tr>
<tr><td><code>State consistency</code></td><td>Critical vehicle states related to sensors, flight mode, pre-arm check, and flight stage</td><td colspan="2">gyroscope, accelerometer, magnetometer, barometer, GPS, parachute, pre-arm check, flight mode, flight stage</td></tr>
</table>

Table 2: Summary of how PatchVerif obtains the invariant property features from flight logs on ArduPilot and PX4.

3. The patch has been committed in the last two years.

We refer to these patches as `TestSet`. Section 9 provides details on how many patches we cannot trigger on the simulator.

8.2 Training Results

We train SVMs in PatchVerif's bug oracle separately for ArduPilot and PX4 through `TrainSet`$_{\mathtt{F}}$ (i.e., faulty patches) and `TrainSet`$_{\mathtt{C}}$ (i.e., correct patches). To train the SVMs, we need to run a mission and then extract the RV's states as features (Section 6.5.2). We leverage an identical mission to obtain features by running an RV control program that includes a patch from `TrainSet`$_{\mathtt{C}}$. On the contrary, each faulty patch from `TrainSet`$_{\mathtt{F}}$ requires us to execute a different set of missions (i.e., inputs) to trigger it. We previously obtained such inputs, when generating `TrainSet`$_{\mathtt{F}}$.

We obtain optimal parameters for the SVMs by running 5-fold cross-validation. To evaluate the trained SVM models, we shuffle the order of the training data (i.e., `TrainSet`$_{\mathtt{F}}$ and `TrainSet`$_{\mathtt{C}}$) and run 5-fold cross-validation. We repeat the shuffling and cross-validation $1,000$ times to obtain the average inference accuracy. As a result, PatchVerif achieves 97.3% precision, 91.2% recall, and 94.1% F1-score in ArduPilot. PatchVerif shows 98.5% precision, 93% recall, and 95.7% F1-score in PX4.

8.3 Previously Unknown Bugs and Responsible Disclosure

We used PatchVerif to analyze the $1,000$ patches in the `TestSet`. Specifically, we tested each patch in `TestSet` until PatchVerif could not detect any increment in source code coverage, for a maximum mutation time of 30 minutes per patch. As a result, PatchVerif classified 117 patches as faulty patches.

We reported these 117 faulty patches to the developers of ArduPilot and PX4. From the developers' responses, we considered 2 of them as false positives (while PatchVerif classifies them as faulty, they are actually correct patches). We will detail these false positives in Section 8.5. Among the other 115 patches, as shown in Table 3, 103 of them have been acknowledged by developers as faulty, and we are currently awaiting an answer for the other 12. Among the 103 acknowledged faulty patches, the bugs introduced by 51 of them have been currently fixed, and the developers are fixing additional 9 bugs at the time of writing. For the other 43 patches, 38 are memory bugs, and 5 cases are simulator bugs. The developers did not consider the other 43 bugs as urgent issues, though they admitted their presence.

We notice that, in theory, some of the 115 identified faulty patches could have already been fixed by a subsequent patch. However, we have never encountered such a situation.

Regarding false negatives, two authors of this paper manually analyzed each patch in the `TestSet` and found 6 missed faulty patches, which we consider as false negatives (more details are provided in Section 8.5).

We believe that our bug reporting did not require unreasonable effort by developers for the following reasons. (1) Before reporting every 117 faulty patches, we manually checked them and manually wrote an appropriate detailed description in the corresponding bug report, e.g., bug-triggering inputs, flight log data, and physical effects of buggy behaviors. (2) 103 out of 117 reported faulty patches have been currently acknowledged by developers. (3) The faulty patches were not reported all at once, but over several months.

8.4 Analysis of the Discovered Bugs

Bug Types. We divide the found faulty patches into three types. (1) "Incomplete fixes" are patches that fix a bug only partially, hence a buggy behavior can still occur. (2) "Incorrect bug fixes" are patches that fix a bug but break an existing functionality. (3) "Incorrect new feature implementations" are patches that implement a new feature but break an existing functionality. PatchVerif discovers 3 (2.6%) "Incomplete fixes", 62 (53.9%) "Incorrect bug fixes", and 50 (43.5%) "Incorrect new feature implementations" patches (Table 3).

Further, we classify the 115 patches according to whether they cause logic bugs or memory bugs. We found that 60 (52.2%) cases are memory bugs, and 55 (47.8%) cases are logic bugs. Developers have acknowledged 59 (98.3%) of the memory bugs and 44 (80%) of the logic bugs. Out of the admitted bugs, 13 (23.2%) of the memory bugs have been fixed, and 38 (90.5%) of the logic bugs have been patched.

Physical Effects of Bugs. As shown in the Table 3 "Physical effects of buggy codes" column, we group the bugs' impacts into three categories. (1) "Unstable attitude/position control"

RV control software	# of bugs	# of acks	# of fixed bugs	# of bugs being fixed	Bug type			Physical effects of buggy codes		
					Incomplete fixes	Incorrect bug fixes	Incorrect new feature implementations	Unstable attitude/ position control	Fail to finish a mission	Crash into ground
ArduPilot	100	90	45	3	3	48	49	29	2	69
PX4	15	13	6	6	0	14	1	7	0	8
Total	115	103	51	9	3	62	50	36	2	77

Table 3: Summary of found 115 previously unknown bugs introduced by faulty patches in ArduPilot and PX4.

represents the vehicle losing its attitude or position control. (2) "Fail to finish a mission" means that the vehicle is stuck in the same location and fails to reach a final waypoint. (3) "Crash into ground" happens when the vehicle loses attitude control and then it goes into free fall. Out of the 115 bugs, 36 bugs (31.3%) cause "Unstable attitude/position control", 2 bugs (1.7%) lead to "Fail to finish a mission", and 77 bugs (67%) result in "Crash into ground".

Reproducing Bugs on a Real Drone. The faulty patches were found using a simulator. To confirm if the bugs they cause can be reproduced on a real vehicle, we manually inspected the found bugs' code location. In case we found the buggy code included in the firmware, we assumed that the bug can be reproduced on the vehicle. To perform this experiment, we used a Pixhawk 4 flight controller, and we confirmed that 74 out of the 115 bugs can be reproduced on a real vehicle. Other 41 bugs only occur if a vehicle manufacturer sets some configuration parameters in specific ways. Yet, they are not reproducible in the vehicle we used for testing, since it uses a different set of parameters.

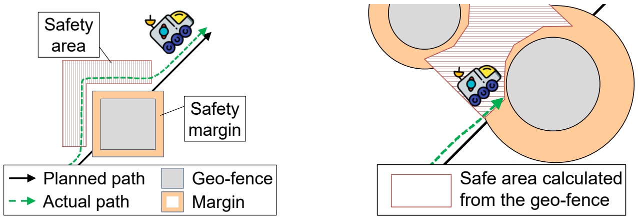
8.5 False Negatives and False Positives

False Negatives. The false negatives represent patches that, while PATCHVERIF classifies them as correct, they are actually faulty patches. PATCHVERIF produced 6 false negatives for the following reasons. First, some patches, while being faulty, do not impact the RV's physical behaviors, but they impact other aspects, such as how messages are shown on the RV's display. Second, PATCHVERIF fails to detect bugs related to RV software's supplementary features, such as logging the flight history to an SD card. PATCHVERIF focuses only on features that impact an RV's navigation, hence it cannot detect such cases. To address both cases, a user may add other features to the invariants considered by PATCHVERIF.

False Positives. False positives are cases in which PATCHVERIF classifies a patch as faulty, while, in reality, it is correct. PATCHVERIF produced 2 false positives when analyzing the `TestSet` dataset.

In one of them, the patch makes a sailboat fail to stay in a constant position, when in `LOITER` mode. On the contrary, the sailboat remains in a fixed location when using the unpatched version of the software. We reported this bug to developers. They answered that the sailboat inevitably fails to stay in a constant position because it does not have a motor to overcome the water current. Yet, the simulator incorrectly simulated the sailboat.

In the other false positive case, PATCHVERIF detected a large position error in the patched version, when the drone follows a mission containing a mix of spline and straight waypoints. Yet, this behavioral difference was intended. In fact, ArduPilot developers decided to patch the software to change the RV behavior in the case of missions consisting of mixed types of waypoints. This patch results, by design, in the RV following spline waypoints less accurately.



(a) A polygon geo-fence. (b) Two circular geo-fences.

Figure 6: When circular geo-fences are set, in subfigure (b), the safe area (red color) and safety margin area (orange color) overlap each other. This overlapping causes the RV's incorrect behavior.

Listing 3 A faulty patch [44] creates safe areas from fences.

```
+// Create safe-zone near from circular geo-fences
+bool AP_OADijkstra::create_polygon_fence_visgraph() {
+  if (!expand_to(num_circles * num_points_per_circle)) {
+    // Raise an out of memory error }
```

8.6 Case Studies

We now present two previously unknown faulty patches found by PATCHVERIF.

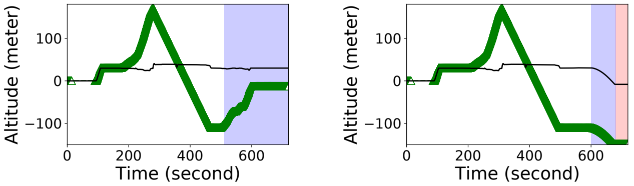
8.6.1 Case Study 1: Object Avoidance Failure

ArduPilot leverages two different object avoidance algorithms at the same time: Dijkstra's path planning [16] and "simple avoidance" algorithm [58]. These algorithms are used both to avoid obstacles and to avoid "geo-fences"[2], as shown in Figure 6a.

To find a path to a destination without any collision, the Dijkstra's path planning algorithm [16] creates safe areas around any object or geo-fenced location. In doing so, it first considers a safety margin around the to-be-avoided area, then, outside this safe margin, it determines a safety area. Later, in computing the shortest path, the Dijkstra's path planning algorithm takes care of generating a path that remains inside the safe areas, while avoiding any obstacle (or geo-fenced area) and any corresponding safety-margin area.

Yet, Dijkstra's path planning could fail to find a path due to its high computational complexity when the vehicle meets a fast-moving object (e.g., another vehicle). To handle these cases, ArduPilot also supports a "simple avoidance" algo-

[2] A geo-fence is an area in which RVs are forbidden from entering.



(a) Normal behaviors of the terrain-following.

(b) Crashing on the terrain after deploying a faulty patch [67].

Figure 7: Altitude values in different versions of software.

rithm [58] to either stop the vehicle or go backward if the vehicle enters a safety margin area. In a sense, the "simple avoidance" algorithm acts as an emergency stop, in case the Dijkstra's path planning algorithm gets "too close" to (i.e., within the safety margin of) a geo-fenced area.

Identified Issue. ArduPilot developers created a patch [44] to support a new type of geo-fences. This new type of geo-fences has a round shape, as shown in Listing 3. During its analysis, PATCHVERIF created multiple circular geo-fences overlapping with a planned path to trigger the patch code from line 4 in Listing 3. However, creating multiple circular geo-fences near the planned path triggers a bug in Dijkstra's path planning.

Specifically, the bug is due to the fact that in the presence of circular geo-fences the Dijkstra's path planning algorithm creates a safety area *overlapping* with safety margin areas, as shown in Figure 6b. This incorrect behavior causes the "simple avoidance" algorithm to be triggered when the vehicle enters the safety margin area. In turn, this causes the RV to move backward. However, since the RV is also within the safety area, it will then start moving forward again. Therefore, the RV starts to repeatedly move back and forth near the board of a margin area, and it is unable to complete its mission. To fix this bug, the Dijkstra's path planning algorithm must not allow the margin area and the safe zone to overlap. We reported this faulty patch and its corresponding bug to ArduPilot developers, and they acknowledged them. In their response, they mentioned that ArduPilot will provide users with configuration parameters to adjust safe areas near circular geo-fences.

Attack. An attacker can exploit this bug by putting circular-shaped objects on the victim RV's path. This makes the RV stuck in an attacker-controlled location. We note that this attack is stealthy since the attacker does not need to send any explicitly malicious input to the RV, but only needs to change environmental conditions.

8.6.2 Case Study 2: Terrain-Following Failure

When a drone flies over a curved terrain, the terrain-following feature [65] allows the drone to maintain a specified distance above the terrain using LiDAR or ultrasound sensors. For example, the green line in Figure 7a shows the calculated altitude, with respect to the take-off altitude, while the black line represents the altitude with respect to the terrain below the drone. In the example, to maintain approximately 30 meters of vertical separation between itself and the terrain below, the drone climbs between 200 and 300 seconds and descends between 300 and 500 seconds (green line), since in that timeframe it encountered and surpassed a hill.

Identified Issue. The developers discovered a bug in the terrain-following feature. Specifically, when a drone is flying over steep terrain, the vehicle does not slow down its horizontal speed, making the drone target a much higher altitude than it originally intended. To fix this issue, the developers created a patch [66] that makes the drone decrease its horizontal speed while climbing. However, PATCHVERIF disclosed that this patch is buggy. In fact, this patch makes the drone subtract an offset value from the current altitude every time it switches to the terrain-following mode. Yet, this patch does not remove such offset when this mode is disabled. Therefore, if a drone switches to the terrain-following mode multiple times, the offset keeps adding up.

Figure 7b shows an example of such behavior. In the example, the white color represents the timeframe in which the drone is navigating towards the first waypoint while turning on the terrain-following. PATCHVERIF then switches its current flight mode to altitude hold mode after the drone arrives at the first waypoint, and immediately switches to navigating to a second waypoint, restarting the terrain-following mode (the blue color background represents the timeframe in which the drone navigates to the second waypoint). However, the drone subtracts an offset value from the current altitude twice, causing it to target an altitude that is too low (in the timeframe colored in red in Figure 7b). As a result, the drone eventually crashes into the ground. We reported this bug to ArduPilot developers, and they consequently fixed it.

Attack. An attacker, who is capable of changing the drone's flight mode, can trigger this bug multiple times, making the drone eventually crash on the terrain. We note that changing the drone's flight mode does not look as a malicious behavior for an external observer. In fact, (*i*) RV control software often automatically changes its flight mode as a fail-safe mechanism [18, 70], and (*ii*) an attacker only needs to switch to the follow-terrain mode for a very brief time and can then return to the previous flight mode.

8.7 Time Requirements

We measure the time required by each step of PATCHVERIF. To this aim, we randomly selected 80 patches from our dataset in Section 8.1 to measure the time spent on each step. Each step in Figure 3 takes, on average: less than 1 minute to analyze the patch's physical impact (❷), 17.5 minutes to find patch-triggering inputs (❸), 8.1 minutes for the symbolic execution (❹), 5.3 minutes to infer the patch type (❺), and 30 minutes to mutate test cases (❻).

8.8 Ablation Study

To evaluate each component's contributions, we conduct an ablation study as summarized in Table 4. We run ablation ex-

	Ablation settings									
ID	Infer physical impact of patches (❷)	Infer inputs related to patches (❸)	Symbolic execution (❹)	Analyze patch type (❺)	Run simulators (6a)	Bug oracle (6b)	Mutate test cases (6c)	Evaluation purpose	FN	FP
AB$_1$	✗	✗	✗	✓	✓	✓	✓	Effect of input selection	59	1
AB$_2$	✓	✓	✗	✓	✓	✓	✓	Symbolic execution	8	1
AB$_3$	✓	✓	✓	✗	✓	✓	✓	Patch type analyzer	11	6
AB$_4$	✓	✓	✓	✓	✓	✗	✓	Five physical invariants	24	14
AB$_5$	✓	✓	✓	✓	✓	✓	✗	Mutation algorithm	37	1

Table 4: Summary of ablation settings and results, used to evaluate each component's contribution. FN and FP denote false negatives and false positives, respectively. For this experiment, we used a dataset of 80 faulty patches (TrainSet$_{\text{F}}$) and 80 correct patches. On this dataset, the full PatchVerif pipeline resulted in 7 FN and 1 FP. AB$_4$'s FN and FP are counted when it uses only the Timeliness invariant.

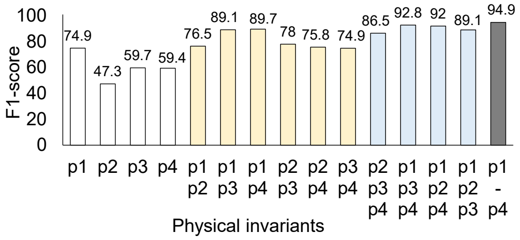


Figure 8: Ablation study of the physical invariants. p1, p2, p3, and p4 represent Timeliness, Precise navigation, Stability, and Efficiency, respectively. We employ SVM classifiers.

periments on 160 patches: (*i*) 80 faulty patches (TrainSet$_{\text{F}}$) and (*ii*) 80 correct patches (TrainSet$_{\text{C}}$) in Section 8.1.

First, we exclude components ❷-❹ in Figure 3. We refer to the version of PatchVerif excluding ❷-❹ as AB$_1$ in Table 4. These components infer the physical impact of the patch (e.g., a patch affects the RV's behaviors during CIRCLE flight mode) and they find inputs that are likely triggering the patch code snippet (e.g., inputs to trigger the CIRCLE flight mode). We run AB$_1$ for a maximum of one hour for each patch. As a result, AB$_1$ finds only 21 out of 80 faulty patches. We note that these 21 faulty patches do not require any input mutation to trigger the buggy behaviors. On the contrary, PatchVerif discovers all the faulty patches. These results prove that reducing the huge input space based on the patch code and mutating inputs only related to the patch significantly improves the time spent to trigger buggy behaviors.

Second, we exclude the symbolic execution (❹ in Figure 3), which enables PatchVerif to trigger unreachable branches in patch code snippets, from PatchVerif (AB$_2$ in Table 4). AB$_2$ fails to fully test all branches of 16 out of the 160 patches. Yet, AB$_2$ still triggers all buggy behaviors except for one faulty patch. The result shows that most of the buggy behaviors caused by faulty patches can be triggered without symbolic execution. We believe that triggering unreachable branches is still required because it enables PatchVerif to discover (*i*) an additional faulty patch in this ablation study and (*ii*) six additional bugs, as explained in Section 6.3.

Third, we exclude the patch type analyzer (❺ in Figure 3). We call it AB$_3$ in Table 4. This modification forces that AB$_3$'s bug oracle always leverages all of the five physical invariants. Compared to PatchVerif, AB$_3$ produces additional five false positives (i.e., the patch is classified as a faulty patch since it violates an invariant, but it is actually not a faulty patch) and four false negatives (i.e., while AB$_3$ classifies them as correct, they are actually faulty patches). The reason for the five false positives is that correct patches may naturally violate some of the five physical invariants. For instance, a patch, which implements a delay command, increases the time required to finish a mission and battery consumption. Yet, the patch does not contain any buggy code. Regarding the four false negatives, the main reason is that faulty patches related to the RV's ground speed decrease the mission completion time but increase position errors. Leveraging all physical invariants neutralizes the faulty patches' side effects if the patches change the RV's ground speed. For example, a faulty patch in Section 3 makes a rover fail to decrease its speed near a corner. The faulty patch decreases the time to finish the mission (due to maintaining a high speed) but increases deviations from a planned path in Figure 2a. These results show that we need to selectively consider the five physical invariants to properly evaluate the effects of a patch.

Fourth, the bug oracle (five physical invariants) is mandatory for running PatchVerif. Instead of excluding the whole bug oracle (6b in Figure 3), we exclude each physical invariant from the bug oracle (we call it AB$_4$) and measure its performance. As shown in Figure 8, leveraging only one physical invariant achieves at most a 74.9% F1-score. On the contrary, when AB$_4$ uses all the five physical invariants, it shows the best F1-score (94.9%). Such results prove that we need to leverage multiple physical invariants to detect various symptoms of the buggy behaviors caused by faulty patches.

Lastly, we exclude the mutation algorithm (6c in Figure 3). We call it AB$_5$ in Table 4. AB$_5$ randomly selects inputs and input assignments without any mutation guidance, e.g., target_altitude = random(). We note that AB$_5$ still leverages the test case template (i.e., a default mission plan), although we exclude input mutation logic. This is because a purely random order of inputs without any mission template cannot even make the RV take-off from the ground (e.g., mutating inputs to increase altitude before the take-off flight stage). We set one hour as the maximum time out to trigger buggy behaviors created by faulty patches. AB$_5$ discovers 43 out of the 80 faulty patches. 41 out of the 43 detected faulty patches do not require any input mutation to trigger the buggy behaviors or are easily triggered by changing the RV's flight stages. We note that $\mathtt{AB}_5$ wastes significant time running self-sabotaging missions, e.g., decreasing motor speed and crashing on the ground. In these cases, we cannot detect any buggy behaviors since both the unpatched and patched RV software fail to conduct a test case at the same time. On the contrary, PATCHVERIF finds all 80 faulty patches because its mutation algorithm excludes input assignments leading to self-sabotaging behaviors and it guides the input assignments based on distances calculated from physical invariants.

8.9 Comparison with Other Works

Some fuzzing approaches have been previously proposed to detect bugs in RVs [30, 32]. While none of the previous approaches focuses specifically on the analysis of patches (like PATCHVERIF does), we evaluated whether they could still detect (some of) the bugs PATCHVERIF discovered.

RVFuzzer [32] mutates configuration parameters and wind to detect control instability bugs (e.g., deviation from a planned path or unstable attitude). RVFuzzer cannot discover 88 out of the 115 bugs identified by PATCHVERIF. 66 out of 88 missed bugs, to be triggered, require specific combinations of user commands, configuration parameters, and environmental settings. RVFuzzer cannot generate such combinations since it only focuses on configuration parameters and wind, i.e., a limitation of its input mutation. 1 out of 88 missed bugs is caused by a limitation of its bug oracle that does not consider *mission completion time*, *battery consumption*, and *unexpected changes in RV states*. 21 out of 88 missed bugs are due to limitations of the input mutation and bug oracle.

PGFuzz [30], on the other hand, mutates all three types of inputs, based on temporal logic formulas that define RVs' expected behaviors. Yet, PGFuzz still cannot find 53 out of the 115 bugs found by PATCHVERIF. PGFuzz has two main limitations that prevent it from discovering these 53 bugs. The first limitation is in its input mutation, where PGFuzz does not directly test a patch's correctness as it mutates inputs only based on the provided formulas. Thus, PGFuzz does not test the environmental conditions that, while relevant to test a patch, are not directly related to the tested formula. The second limitation is in its bug oracle. PGFuzz uses manually-written temporal logic formulas. However, these formulas are too general to detect some of the incorrect behavioral differences introduced by faulty patches. In theory, PGFuzz's users could add formulas, allowing it to detect more bugs. However, this requires them to manually write these formulas, based on an already-known, precise notion of what distinguishes a correct behavior from a faulty one. Out of the 53 missed bugs, 2 of them are missed only because of PGFuzz's input mutation, 32 of them are missed only due to its bug oracle, and 19 of them are missed because of both of these limitations.

Further, PATCHVERIF discovers all faulty patches identified by RVFuzzer, but it fails to discover 3 out of 156 bugs reported by PGFuzz. These three bugs make ArduPilot incorrectly set the camera angles. However, PATCHVERIF focuses only on features that impact an RV's navigation, hence it cannot detect faulty patches related to such supplementary features (Section 8.5). To detect the three cases, a user may add other features to the invariants considered by PATCHVERIF.

9 Limitations and Discussion

Code Coverage on RV Simulators. PATCHVERIF runs test cases on simulators to discover faulty patches. The simulators cannot guarantee full source code coverage. For instance, test suites of ArduPilot and PX4 [7, 52] achieve 56% and 40% of code coverage on the simulators at the time of writing because the device driver code cannot be run on the simulators [3]. To tackle this issue, we might leverage Hardware-in-the-loop (HIL) simulation [5,49] and Simulation-In-Hardware (SIH) [51] where firmware is run on real hardware. Yet, we believe that running HIL and SIH to test patches is not practical because they require numerous hardware devices to test all hardware configurations.

To analyze the impact of this limitation, we aim to answer the following two research questions: (*RQ1*) How many patches PATCHVERIF cannot test because they are in device drivers? (*RQ2*) Are there any patches that are not in device drivers and cannot be tested by PATCHVERIF?

To obtain $1,000$ patches (`TestSet`) in Section 8.1, we initially collected a total of $1,137$ patches. The missing 137 patches update code in device drivers. For this reason, they are not executed by simulators, and, consequently, they cannot be tested by PATCHVERIF (*RQ1*). Regarding (*RQ2*), we could not find any patch that is not in device drivers and that cannot be triggered by PATCHVERIF. These results show that (*i*) developers actively create patches for changing RV control logic, which is not related to device drivers, and (*ii*) PATCHVERIF can test most of the patches (87.9%) despite achieving limited code coverage on ArduPilot and PX4, respectively.

Correctness of the Unpatched Version. PATCHVERIF compares an unpatched version with a patched one to detect bugs. Here, the unpatched version can be a released version or the version just before applying the patch (Section 6.5.1). PATCHVERIF's detection accuracy depends on the unpatched version. If the unpatched version initially violates one of the five invariant properties, PATCHVERIF cannot detect an incorrect behavior of the patched version. In our evaluation (Section 8), we could not find any unpatched version that makes PATCHVERIF fail to detect faulty patches because we mainly leverage the released versions for the faulty patches in our dataset. Yet, we cannot guarantee that the version just before applying the patch does not contain any bug that might lead to incorrect behaviors. This is because, unlike the released version, the unpatched version is not heavily tested.

Limitations of the Five Physical Invariants. PATCHVERIF's five physical invariants can detect many of RV's abnormal physical behaviors (e.g., attitude and position issues). While we think that these five invariants cover the majority of the abnormal physical behaviors potentially caused by faulty patches (as explained in Section 5), it is possible that some faulty patches can only be detected using additional invariants. Indeed, we are aware that 6 faulty patches in Section 8.5, within the TestSet dataset, cannot be detected with our current five invariants. To address this issue, a user could potentially add other invariants to be considered by PATCHVERIF. However, this task requires some manual effort.

Porting PATCHVERIF to Other RVs. PATCHVERIF requires a simulator, which is commonly available for open-source RV control software [21, 59]. Further, we assume that other RV software leverage MAVLink [42] as their communication protocol between the RV software and ground control stations because MAVLink is a de facto standard. Users can port PATCHVERIF to the other RV control software through the following steps: (1) collecting faulty and correct patches, (2) identifying features as physical invariants from the correct and faulty patches, (3) constructing a list of environmental factors from documentation, and (4) training PATCHVERIF's bug oracle from the collected patches. We believe that porting PATCHVERIF to a new RV software is not a burden, given that users of PATCHVERIF are RV developers who are already familiar with RV software and simulators. Two authors spent around 12 hours on these manual porting tasks.

Security-relevance of Exploiting Logic Bugs. Exploiting faulty patches is attractive for attackers since it can be achieved even when an attacker has partial control over a drone's inputs (e.g., an attacker has control only over environmental conditions, but not over user commands). For the 74 faulty patches reproduced on a real drone (Section 8.4), we analyze the input types (i.e., user commands, configuration parameters, and environmental conditions) the attacker needs to control to trigger faulty patches. Our analysis reveals that an adversary who controls a single input type can trigger 67 out of the 74 faulty patches (90.5%) discovered by PATCHVERIF.

An attacker can exploit faulty patches to cause physical consequences that are otherwise difficult to achieve. For instance, the faulty patch in Section 8.6.1 enables attackers to stealthily cause a failure in the object avoidance module, by placing, for example, circular-shaped objects on the victim RV's path (i.e., changing environmental conditions).

Faulty patches enable exploiting bugs for a long time without getting detected. For example, the faulty patch in Section 8.6.2 makes the RV target an incorrect altitude and eventually crash. An attacker can trigger this faulty behavior by changing the victim RV's flight mode. In turn, changing the flight mode is achievable by jamming GPS signals. In fact, the RV software changes its flight mode (due to a fail-safe logic) when the GPS signal is lost. Since GPS signal loss is not an obvious symptom of any attack, as GPS glitches frequently occur in highly urbanized areas [25], exploiting this faulty patch results in an attack that is hard to detect.

Comparison of Name-based Matching and Taint Analysis. The name-matching method and taint analysis match with, on average, 3.3 and 35.8 physical states/environmental conditions, respectively. We run dynamic analysis (Section 6.2 and Section 6.3) based on the results of each static analysis to measure how much time dynamic analysis spends on finding a set of inputs triggering the patch code. We set one hour as the maximum time out to find the inputs for each patch. We conduct this experiment on randomly selected (*i*) 40 faulty patches from $\texttt{TrainSet}_\texttt{F}$ and (*ii*) 40 correct patches from $\texttt{TrainSet}_\texttt{C}$ in Section 8.1.

As a result, dynamic analysis based on the results of the name-matching method triggers all 80 patches on average in 17.5 minutes. On the contrary, dynamic analysis based on the results of the taint analysis succeeds in triggering 62 out of 80 patches on average in 38.4 minutes. The main reason is that taint analysis detects all indirect data propagation and produces over-approximated results (i.e., over-tainting), which makes dynamic analysis waste most of the time to mutate inputs unrelated to the patch code. We detail the implementation of the taint analysis in Appendix C.

10 Related Work

Invariants-based Patch Validation. Previous works detect faulty patches through program invariants [20, 72]. Correctly patched software does not break the program invariants, while faulty patches make the software violate the invariants. These works extract invariants over variables (e.g., $\texttt{variable}_\texttt{i}$ must be smaller than $\texttt{variable}_\texttt{j}$). Yet, the accuracy of the inferred invariants depends on the completeness of test suites. Further, the invariants in the source code level could not detect logic bugs in RVs because the bugs in RV software can occur regardless of invariants over variables. On the contrary, we extract RVs' physical invariants (e.g., battery consumption) to detect buggy behaviors caused by faulty patches.

Test Case Generation-based Patch Validation. Many existing works [27, 34, 64, 71] attempt to discover faulty patches through test case generation. These works find test inputs that make unpatched and patched programs generate different outputs. If the programs' outputs are not the same as that of a bug oracle, it is regarded as a faulty patch. Yet, these works are not proper to discover bugs in RVs because (*i*) they do not consider finding environmental conditions as test inputs. The faulty patches in RVs only take effect under specific environmental conditions (e.g., terrain and navigation speed), and (*ii*) they assume that a bug oracle exists. However, such an assumption cannot hold in RVs as explained in Section 1.

Forced Execution. Forced execution (such as T-Fuzz [45]) can trigger unreachable branches in patches. T-Fuzz negates a conditional statement in an "if statement" to trigger an unreachable false/true branch, e.g., T-Fuzz converts the predicate from $\texttt{if(a == 0x1234)}$ to $\texttt{if(a != 0x1234)}$. However, we notice that simply negating conditional statements in the RV control software leads to losing context to test the RV's behaviors. For example, the true branch of the following code assumes that `user_input` is not `null`:

if(user_input! = null){target_roll = user_input}. Yet, forced execution cannot change the RV's physical states even if it executes the true branch because the variable (user_input) does not have any meaningful concrete assignment (i.e., user_input is null). On the contrary, PATCHVERIF inserts concrete assignments into the patch to test the RV's behaviors, as explained in Section 6.3.

Symbolic Execution. PATCHVERIF's use of symbolic execution is inspired by existing works [41, 43, 61]. For example, Driller [61] uses symbolic execution to find inputs for conditions that fuzzers cannot satisfy and execute deeper paths in software. However, these works [41,43,61] focus on memory corruption bugs. To discover logic bugs in RV software, PATCHVERIF uses additional techniques, such as the five physical invariants as an oracle and the distance metrics based on the physical invariants as mutation guidance.

Shadow Execution. Comparing unpatched and patched versions to identify faulty patches in general software has also been studied [35, 56]. Ramos et al. [56] detects bugs if a patched version leads to a program crash, but an unpatched version does not cause any program crash. Yet, PATCHVERIF aims to discover logic bugs that do not lead to any program crash but cause unexpected physical behaviors. Kuchta et al. [35] generates test cases exercising the divergence and comprehensively test the new behaviors of the patched version. They assume that they already have a test input that executes the patch code. However, as explained in Section 8.8, our ablation study shows that finding such a patch-triggering input is challenging and that PATCHVERIF's combination of static and dynamic analyses is efficient in discovering the patch-triggering input.

Taint Analysis. Existing taint analysis methods cannot be directly applied to RV control software. For example, Luo et al. [39] tracks how the whole app-level information propagates to Android Framework variables. Their taint analysis is for specific taint sources (unique identifications of an app) and sinks (*get* functions of a data structure that stores the specific app's information). However, taint sinks in RV software are not obvious due to high inter-dependency among variables. To address this challenge, PATCHVERIF uses the name-based matching, as explained in Section 6.1.

11 Conclusions

We introduce PATCHVERIF, a patch verification framework for RVs. PATCHVERIF has three key aspects that distinguish it from other methods: (*i*) leveraging physical invariants as a bug oracle to detect symptoms of faulty patches, (*ii*) testing both the behaviors affected by a patch and environmental conditions that affect the patch, and (*iii*) using coverage and physical invariants-based guidance to mutate inputs that make the patched software violate the invariants if the patch is faulty. PATCHVERIF found 115 previously unknown bugs introduced by faulty patches, 103 of which have been acknowledged, and 51 bugs have been fixed.

Acknowledgments

We thank the anonymous reviewers for their invaluable suggestions. 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] AMD performance regression. https://tinyurl.com/whhsuaca, 2022.

[2] ArduPilot Coding Style Guide. https://tinyurl.com/yjcjx7up, 2022.

[3] ArduPilot code coverage. https://coveralls.io/github/ArduPilot/ardupilot, 2022.

[4] ArduPilot environmental factors. https://tinyurl.com/4yjj99s8, 2022.

[5] ArduPilot Hardware in the Loop Simulation (HITL). https://tinyurl.com/53h8chs3, 2022.

[6] ArduPilot log. https://tinyurl.com/exya4r75, 2022.

[7] Autotest. https://tinyurl.com/6ampumny, 2022.

[8] Bad Windows Update. https://tinyurl.com/8evh9rs2, 2022.

[9] Boeing 737 max crash. https://tinyurl.com/nenryb9p, 2022.

[10] Marcel Böhme and Abhik Roychoudhury. Corebench: Studying complexity of regression errors. In *Proceedings of the international symposium on software testing and analysis*, 2014.

[11] Tim Chen, Leonid I Ananiev, and Alexander V Tikhonov. Keeping kernel performance from regressions. In *Proceedings of the Linux Symposium*, 2007.

[12] Hongjun Choi, Sayali Kate, Yousra Aafer, Xiangyu Zhang, and Dongyan 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.

[13] Hongjun Choi, Wen-Chuan Lee, Yousra Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, Dongyan Xu, and Xinyan Deng. Detecting attacks against robotic vehicles: A control invariant approach. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[14] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 1995.

[15] Delay command. https://tinyurl.com/t7xp9298, 2022.

[16] Dijkstra's path planning. https://tinyurl.com/2habdyx3, 2022.

[17] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. Automating patching of vulnerable open-source software versions in application binaries. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[18] EKF fail-safe. https://tinyurl.com/y2e44bp6, 2022.

[19] Emergency stop is broken. https://tinyurl.com/4kuxccx9, 2022.

[20] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE transactions on software engineering*, 2001.

[21] Gazebo. http://gazebosim.org/, 2022.

[22] Gcov. https://tinyurl.com/8knsh8kc, 2022.

[23] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 2014.

[24] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 1997.

[25] Li-Ta Hsu. GNSS multipath detection using a machine learning approach. In *Proceedings of the 20th International Conference on Intelligent Transportation Systems (ITSC)*, 2017.

[26] Initialize rngOnGnd. https://tinyurl.com/2p8cfbbw, 2022.

[27] Wei Jin, Alessandro Orso, and Tao Xie. Automated behavioral regression testing. In *Proceedings of the 3rd international conference on software testing, verification and validation*, 2010.

[28] Chijung Jung, Ali Ahad, Yuseok Jeon, and Yonghwi Kwon. SWARMFLAWFINDER: Discovering and Exploiting Logic Flaws of Swarm Algorithms. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2022.

[29] Chijung Jung, Ali Ahad, Jinho Jung, Sebastian Elbaum, and Yonghwi Kwon. Swarmbug: debugging configuration bugs in swarm robotics. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[30] Hyungsub Kim, Muslum Ozgur Ozmen, Antonio Bianchi, Z. Berkay Celik, and Dongyan Xu. PGFUZZ: Policy-guided fuzzing for robotic vehicles. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2021.

[31] Hyungsub Kim, Muslum Ozgur Ozmen, Z Berkay Celik, Antonio Bianchi, and Dongyan Xu. PGPATCH: Policy-Guided Logic Bug Patching for Robotic Vehicles. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2022.

[32] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. RVFUZZER: Finding input validation bugs in robotic vehicles through control-guided testing. In *Proceedings of the USENIX Security Symposium*, 2019.

[33] KLEE. https://klee.github.io, 2022.

[34] Bogdan Korel and Ali M Al-Yami. Automated regression test generation. *ACM SIGSOFT Software Engineering Notes*, 1998.

[35] Tomasz Kuchta, Hristina Palikareva, and Cristian Cadar. Shadow symbolic execution for testing software patches. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2018.

[36] LCOV. https://github.com/linux-test-project/lcov, 2022.

[37] Wei Le and Shannon D Pattison. Patch verification via multiversion interprocedural control flow graphs. In *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[38] Linux performance regression. https://tinyurl.com/4wwyxext, 2022.

[39] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Peng Liu. System service call-oriented symbolic execution of android framework with applications to vulnerability discovery and exploit generation. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, 2017.

[40] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. Spider: Enabling fast patch propagation in related software repositories. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[41] Paul Dan Marinescu and Cristian Cadar. KATCH: High-coverage testing of software patches. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, 2013.

[42] MAVLink. https://mavlink.io/en/, 2022.

[43] Matthias Neugschwandtner, Paolo Milani Comparetti, Istvan Haller, and Herbert Bos. The borg: Nanoprobing binaries for buffer overreads. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, 2015.

[44] Object avoidance for fences. https://tinyurl.com/kcvy9suc, 2022.

[45] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: Fuzzing by program transformation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[46] Pivot turns. https://tinyurl.com/bkr3bk7p, 2022.

[47] Pivot turns broken. https://tinyurl.com/3wzpa8kt, 2022.

[48] PX4 environmental factors. https://tinyurl.com/4vk3wuct, 2022.

[49] PX4 Hardware in the Loop Simulation (HITL). https://tinyurl.com/tajtakn4, 2022.

[50] PX4 log. https://tinyurl.com/2jax75ct, 2022.

[51] PX4 Simulation-In-Hardware (SIH). https://tinyurl.com/txsypww4, 2022.

[52] PX4 test cases. https://tinyurl.com/4chk8x7a, 2022.

[53] Pymavlink. https://pypi.org/project/pymavlink/, 2022.

[54] Zichao Qi, Fan Long, Sara Achour, and Martin 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*, 2015.

[55] Raul Quinonez, Jairo Giraldo, Luis Salazar, Erick Bauman, Alvaro Cardenas, and Zhiqiang Lin. SAVIOR: Securing autonomous vehicles with robust physical invariants. In *Proceedings of the USENIX Security Symposium*, 2020.

[56] David A Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *Proceedings of the USENIX Security Symposium*, 2015.

[57] scikit-learn Decision Trees. https://tinyurl.com/bdf2y38z, 2022.

[58] Simple object avoidance. https://tinyurl.com/cpzcmcdj, 2022.

[59] SITL. https://tinyurl.com/dvkjwwy9, 2022.

[60] Fix speed nudge. https://tinyurl.com/x9hwzvfx, 2022.

[61] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[62] Philip H Swain and Hans Hauska. The decision tree classifier: Design and potential. *IEEE Transactions on Geoscience Electronics*, 1977.

[63] Shin Hwei Tan and Abhik Roychoudhury. relifix: Automated repair of software regressions. In *Proceedings of the 37th IEEE International Conference on Software Engineering*, 2015.

[64] Kunal Taneja and Tao Xie. DiffGen: Automated regression unit-test generation. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008.

[65] Terrain following. https://tinyurl.com/x2sfkb55, 2022.

[66] Terrain following hits terrain. https://tinyurl.com/y69xsb9t, 2022.

[67] Terrain following initialization. https://tinyurl.com/bpcf3h4d, 2022.

[68] Tesla crash. https://tinyurl.com/jdnv7rzj, 2022.

[69] Tesla on autopilot crash. https://tinyurl.com/57rbbku8, 2022.

[70] Vibration failsafe. https://tinyurl.com/43crr4sw, 2022.

[71] Qi Xin and Steven P Reiss. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017.

[72] Bo Yang and Jinqiu Yang. Exploring the differences between plausible and correct patches at fine-grained level. In *Proceedings of the 2nd International Workshop on Intelligent Bug Fixing (IBF)*, 2020.

[73] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[74] Shahed Zaman, Bram Adams, and Ahmed E Hassan. A qualitative study on performance bugs. In *Proceedings of the 9th IEEE working conference on mining software repositories (MSR)*, 2012.

[75] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. In *Proceedings of the USENIX Security Symposium*, 2018.

[76] Lei Zhou, Fengwei Zhang, Jinghui Liao, Zhengyu Ning, Jidong Xiao, Kevin Leach, Westley Weimer, and Guojun Wang. Kshot: Live kernel patching with smm and sgx. In *Proceedings of the 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020.

A Additional Experiments about the Five Physical Invariants

To confirm the feasibility of detecting faulty patches through the four physical invariants (`Timeliness`, `Precise navigation`, `Stability`, and `State consistency`), we conducted a preliminary experiment on 80 correct and 80 faulty patches in Section 8.1. While performing the preliminary experiment, we decided to choose battery consumption as a new physical invariant (`Efficiency`) for the following reasons. (1) We observe that most correct patches do not increase battery consumption, e.g., 69 out of 80 correct patches ($\texttt{TrainSet}_\texttt{C}$ in Section 8.1) show the same or decreased battery consumption compared to unpatched versions. (2) We notice that most faulty patches degrade the battery efficiency due to unstable attitudes and/or deviated planned path, e.g., 71 out of 80 faulty patches ($\texttt{TrainSet}_\texttt{F}$ in Section 8.1) deplete the RV's battery. (3) Many patches affect battery efficiency, e.g., we found 446 and 382 patches related to the battery performance from ArduPilot and PX4, respectively.

The battery consumption is already related to other invariants, e.g., increasing mission completion time naturally leads to increasing the battery consumption. Yet, it helps PATCHVERIF more clearly distinguish between correct and faulty patches. As shown in Figure 8, when we conduct an ablation study of the physical invariants on the datasets $\texttt{TrainSet}_\texttt{C}$ and $\texttt{TrainSet}_\texttt{F}$ (as defined in Section 8.1), `Efficiency` (p4) enables PATCHVERIF to improve the F1-score from 89.1% to 94.9%.

We note that PATCHVERIF allows RV developers to add additional invariants to PATCHVERIF's bug oracle. Yet, our evaluation (Section 8) already shows that these five invariants are enough to correctly classify most patches.

B Using Other Classification Algorithms

B.1 Decision Trees

We compare SVMs with decision trees (an optimized version of the CART algorithm [57]) for making baseline classification performance and evaluating SVMs' effectiveness. We first find optimal parameters for the decision trees to compare SVMs with the decision trees fairly. We then evaluate the decision trees using the same dataset as SVMs' ones. Particularly, we use 5-fold cross-validation for obtaining optimal parameters for our training dataset and preventing overfitting of the model. "Maximum tree depth" parameter is a limit to stop the further splitting of nodes in a decision tree. We noticed that 3 yields better results than other values in both RV programs. "Alpha" parameter controls how much pruning must be done in the decision tree. We set this parameter to 0.07 in ArduPilot and 0.075 in PX4 since these values show the best accuracy compared to other values.

To evaluate the trained decision tree and SVM models, we shuffle the order of the data and run 5-fold cross-validation. We repeat the shuffling and cross-validation $1,000$ times to obtain average inference accuracy. As a result, the decision tree shows an average 1.8% lower F1-score than SVM's ones.

B.2 Neural Networks

We also compare SVMs with neural networks. Particularly, the neural network consists of seven layers and two denses. "Dropout" is used to prevent the model overfitting by randomly disconnecting the outgoing edges of hidden units specified in the parameter, which we set to 0.2. We leverage "Tanh" as activation function at each layer, "Mse" loss function, "Adam" optimizer, and "Loss" metrics since these parameters yield the best result compared to other parameters. Instead of a constant "Epoch", we use "early stopping", automatically stopping train steps when a monitored metric has stopped improving. Specifically, we leverage "Loss" as the monitoring metric, 10 patience number, and "auto" mode since we noticed that these parameters are the best to reduce the model overfitting. The "Threshold for classification" parameter is for setting a threshold to decide whether a set of features is buggy behaviors or not. We set the threshold to 0.5 because we train the neural network models through a balanced dataset. These neural network models yield similar results to other architectures we have tested. The neural network models show an average 7.2% lower F1-score than SVMs' ones.

We notice that the number of faulty patches in the training dataset is not enough. To tackle this issue, we conduct the following two steps: (*i*) obtaining artificially created bugs from a Generative Adversarial Network (GAN) [23] and (*ii*) training a neural network model [24] through bugs obtained from an RV control software and the GAN model. We create separate GAN and neural network models for each RV control software and leverage them for training and classification.

We first obtain a total of 100 bug cases from the GAN model. We call the artificial bug cases $\texttt{Train}_\texttt{GAN}$. Then, we train a neural network model through 40 correct patches, 40 faulty patches per RV software, and $\texttt{Train}_{\texttt{GAN}}$, which enables PatchVerif to learn how the five physical invariant properties change between correct and faulty patches. These GAN and neural network models yield higher accuracy than other neural network architectures we have experimented with. However, the GAN and neural network models still show an average 5.5% lower F1-score than SVMs.

We note that we have tried using $\texttt{Train}_{\texttt{GAN}}$ for the decision trees and SVMs, but we observed negligible changes in the accuracy of these models. Therefore, we decided not to incorporate $\texttt{Train}_{\texttt{GAN}}$ for them.

B.3 Constant Threshold Oracle

PatchVerif allows RV developers to use another classifier (e.g., constant thresholds), offering flexibility. Unfortunately, ArduPilot and PX4 do not have such thresholds because their test suites only leverage different mission completion times per test case. We obtain the thresholds in ArduPilot using the following steps. (1) We randomly select 64 correct and 64 faulty patches from $\texttt{TrainSet}_{\texttt{F}}$ and $\texttt{TrainSet}_{\texttt{C}}$ (Section 8.1). (2) We calculate a normalized difference in each physical invariant property when we run the same mission on both unpatched and patched software: $\frac{\mathtt{P_i}-\mathtt{P}'_{\mathtt{i}}}{\mathtt{P_i}}$ where $\mathtt{P_i}$ and $\mathtt{P}'_{\mathtt{i}}$ represent i-th physical invariant in an unpatched version and i-th invariant in a patched RV software, respectively. (3) We obtain $\mathtt{f_i}$ by adding up all normalized differences, i.e., $\mathtt{f_i} = \sum_{\mathtt{i}=1}^{\mathtt{n}} \frac{\mathtt{P_i}-\mathtt{P}'_{\mathtt{i}}}{\mathtt{P_i}}$ where $\mathtt{n}$ means the total number of physical invariants, e.g., in case of PatchVerif, the $\mathtt{n}$ is 5 because it uses five physical invariants. (4) We obtain the minimum value $\mathtt{f_{min}}$from a set of $\mathtt{f_i}$. We leverage $\mathtt{f_{min}}$ as the constant threshold to decide whether differences in physical invariant properties are symptoms of faulty patches. (5) We classify other 16 correct and 16 faulty patches from $\texttt{TrainSet}_{\texttt{F}}$ and $\texttt{TrainSet}_{\texttt{C}}$, i.e., these patches are not used for obtaining $\mathtt{f_{min}}$. Then, we repeat the steps (1)-(5) five times, i.e., running 5-fold cross-validation. As a result, using constant thresholds to distinguish the faulty and correct patch classes yields 80.8% precision, 89.5% recall, and 83.7% F1-score. The high F1-score is caused by the fact that correct and faulty classes are significantly different.

Yet, we decided to leverage ML-based classifiers because (*i*) ArduPilot and PX4 do not explicitly mention such constant thresholds in their documentation and (*ii*) some patches require us to distinguish multi-dimensional features which the constant thresholds cannot handle. For example, a correct patch increases the time to finish a mission but decreases position errors.

C Details of Taint Analysis

To implement the static taint analysis, we first construct a data-flow graph (DFG) through the following steps: (1) convert the RV control software into *bitcode* that is the intermediate representation (IR) of LLVM, (2) transform the *bitcode* to its single static assignment (SSA) form, (3) perform Andersen's pointer analysis, which is an inter-procedural, path-insensitive, and flow-sensitive points-to analysis, to handle data propagation through pointers, and (4) construct the DFG that contains all data propagation through scalar and pointer operations. Then, we run the taint analysis on top of the DFG.

Listing 4 An example patch code snippet [26] for illustrating the inter-dependency issue in RV software. Developers add one line of patch code at line 2.

```
1 void NavEKF3_core::InitialiseVariables() {
2   rngOnGnd = 0.05f;
3   ... }
4
5 void NavEKF3_core::EstimateTerrainOffset() {
6   terrainState = MAX (rngOnGnd, ...); ... }
7
8 bool NavEKF3_core::getHeightControlLimit() {
9   height -= terrainState; ...}
10 ...
```

In particular, we assign variables in a patch code snippet as taint sources. Yet, deciding taint sinks in the RV control software is not obvious because variables in the software are highly inter-dependent. This is because the RV software consists of control loops, and almost every variable in the loops refers to variables in other functions. Ideally, inputs to actuators would be taint sinks (i.e., hardware abstraction layer (HAL) of device drivers). However, at this low level, we lose any physical context that the inputs have due to the high inter-dependency among variables. For example, as shown in Listing 4, developers add a simple patch code at line 2. `rngOnGnd` represents the offset at which a range finder sensor is mounted on an RV. `rngOnGnd`'s value is propagated to `terrainState` at line 6 and `height` at line 9. Here, almost every function in the RV software refers to the altitude (i.e., `height`) because RV software's main task is maintaining proper attitude and altitude. When we analyze a data flow of `rngOnGnd`, its value is propagated to 3,227 out of the 7,407 variables (43.6%) in the RV software. Due to the high inter-dependency among variables in the RV software, it might be challenging to perform data-dependency analysis (e.g., taint tracking) [29].

To tackle the high data-dependency, we might consider dynamic analysis without the help of static analysis. Yet, we note that such dynamic analysis needs to find patch-triggering inputs from the whole input space of RVs. Our ablation study ($\mathtt{AB_1}$) in Section 8.8 shows that (*i*) static analysis reduces the huge input space to the input set most likely to trigger a patch, and (*ii*) dynamic analysis without the help of static analysis is unlikely to trigger the patch code because it wastes most of the time to test inputs unrelated to the patch code. Therefore, we assign the synonym table (Figure 4) as the taint sinks. Whenever a new tainted data propagation occurs, we compare the new tainted variable's name with the RV's physical states and environmental conditions on the synonym table.