PatchVerif: Discovering Faulty Patches in Robotic Vehicles

Hyungsub Kim, Muslum Ozgur Ozmen, Z. Berkay Celik, Antonio Bianchi, and Dongyan Xu
Purdue University

USENIX Security Symposium 2023
What are Robotic Vehicles (RVs)?

- Vehicles that move “autonomously” on the ground, in the air, on the sea, under the sea, or in space
What are Faulty Patches?

- Patches unintentionally breaking the software functionality
- Mainly three different types of faulty patches:

1) Partially fixing a buggy behavior
2) Fixing an incorrect behavior but breaking another correct behavior
3) Adding a new feature but introducing a bug
Q: Why are faulty patches important in Robotic Vehicles (RVs)?
Motivation

• Writing patches for RV control software is error prone\(^1\)
  • Developers reverted or fixed 345 faulty patches in ArduPilot and PX4 in the past 5 years

• Faulty patches lead to unwanted physical behavior
  • Mission failure
  • Unstable attitude/position control
  • Crashing on the ground

Q: Why is creating patches for RV control software challenging?

A: Tracking patch-introduced behavioral modifications is difficult.
Pivot Turn (1)

- When a rover is near a corner
  - The vehicle should reduce its speed, turn towards the next waypoint, and continue the navigation.
Pivot Turn (2)

• When a rover is near a corner
  • The vehicle should reduce its speed, turn towards the next waypoint, and continue the navigation.
Motivating Example

void Mode::navigate_to_waypoint() {
    - float desired_speed = g2.wp_nav.get_speed();
    + float desired_speed = g2.wp_nav.get_desired_speed();
}

Returns slower speed while the RV gets near to a waypoint

Returns a constant speed set by a configuration parameter

Developers noticed the buggy behavior only after three months of deploying the faulty patch

This RV can roll overed due to its high speed.

Motivation (4/4)

This RV can roll overed due to its high speed.

Developers noticed the buggy behavior only after three months of deploying the faulty patch

Normal RV behavior before deploying the faulty patch

Abnormal RV behavior after deploying the faulty patch

A faulty patch in a RV control software

Motivation (4/4)
Why do test cases created by developers fail to detect the faulty patch?
Test Cases Created by Developers

- Manually created test cases do not exercise the physical conditions that trigger the buggy behavior.

**Condition 1:** Creating a sharp corner through waypoints

**Waypoint i**

10 m/s

**Condition 2:** Setting a high ground speed (e.g., 10 m/s)
Main Idea of PatchVerif

Let's create test cases based on a given patch!
Overview of PatchVerif

1. Analyze the physical impact of the patch

2. Find inputs triggering the patch

3. Analyze the patch type

4. Mutate test cases

5. Bug oracle

Faulty patches
Analyze Physical Impact of Patches

• We aim to infer
  • An RV’s physical states that are affected by the patch
  • Environmental conditions that affect the patch

```cpp
1 + void AC_Circle::set_center(const Location& _center) {
2 +   if (center.get_alt_frame() == ABOVE_TERRAIN) {
3 +     if (center.get_vector_xy_from_origin(center_xy)) { ... }
4 +   } else { ... }
5 +   else {
6 +     if (!center.get_vector_from_origin(circle_center)) { ... }
```

Step 1:
Extract names of variables and functions in the patch

<A patch implementing terrain-following for the CIRCLE flight mode>
Analyze Physical Impact of Patches

- We aim to infer
  - An RV’s physical states that are affected by the patch
  - Environmental conditions that affect the patch

Step 2: Filter out all but nouns from the variable/function names
Analyze Physical Impact of Patches

• The patch changes
  • The RV’s location, altitude, and flight mode states
• The patch is affected by
  • Terrain environmental factor

Step 3: Match the extracted terms with RV physical states and environmental conditions in the synonym table.

We call these identified states and environments Physical\textsubscript{set}.
Overview of PatchVerif

1. Analyze the physical impact of the patch
2. Find inputs triggering the patch
3. Analyze the patch type
4. Mutate test cases
5. Bug oracle

Faulty patches
Goal: Finding inputs (user commands/configuration parameters) triggering the patch code snippet

Executing inputs related to the identified Physical set:

Physical set: location, altitude, flight mode, terrain

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

<A patch implementing terrain-following for the CIRCLE flight mode>
Overview of PatchVerif

1. Analyze the physical impact of the patch
2. Find inputs triggering the patch
3. Analyze the patch type
4. Mutate test cases
5. Bug oracle

Faulty patches
1) Assign a value greater or lesser than default value to an input (such as ground speed)
2) If it brings a negative impact, PatchVerif keeps increasing/decreasing the input’s value
Mutate Test Cases

- Mutating the identified inputs to test the patch
- Increasing the rover’s speed (5 m/s)

![After deploying the faulty patch](image1)
![Before deploying the faulty patch](image2)

<table>
<thead>
<tr>
<th>Battery consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>W/second</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>After deploying the faulty patch</td>
</tr>
<tr>
<td>3398</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Position error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meter (m)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>After deploying the faulty patch</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>
Bug Oracle

- Mutating the identified inputs to test the patch
- Increasing the rover’s speed (10 m/s)

After deploying the faulty patch

Battery consumption

<table>
<thead>
<tr>
<th></th>
<th>Before deploying the faulty patch</th>
<th>After deploying the faulty patch</th>
</tr>
</thead>
<tbody>
<tr>
<td>W/second</td>
<td>3200</td>
<td>4457</td>
</tr>
</tbody>
</table>

Position error

<table>
<thead>
<tr>
<th></th>
<th>Before deploying the faulty patch</th>
<th>After deploying the faulty patch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meter (m)</td>
<td>5</td>
<td>36</td>
</tr>
</tbody>
</table>
Evaluation Results

• Dataset
  • 1,000 patches
    • We did not know whether they were faulty or correct.

• Results
  • PatchVerif discovered 115 previously-unknown faulty patches
  • 103 bugs have been acknowledged
  • 51 bugs have been patched
A Bug in Dijkstra Object Avoidance Algorithm

Demo: A faulty patch discovered by PatchVerif in ArduPilot's object avoidance with Dijkstra's algorithm
Summary

• Writing patches for RV software is error prone
  • Identifying patch-introduced behavioral modifications is difficult

• PatchVerif
  • Patch profiling
    • Extracting inputs related to a patch

  • Generate new test cases, by mutating patch-related inputs

  • 115 previously-unknown faulty patches
Thank you! Questions?

kim2956@purdue.edu

https://github.com/purseclab/PatchVerif

I will be on the academic job market in Fall 2023
Limitations of Previous Approaches

What about traditional fuzzers (AFL, libFuzzer)? No

• Bug oracle: Memory access violation

What about fuzzers for RVs? No

• Mutation:
  • Do not mutate waypoints
• Bug oracle:
  • Require manually-specified notion of what a “correct behavior” is
Q: Why do we use a name-based matching rather than taint analysis?

A: Over-tainting issues
Physical Invariants as Bug Oracles

• PatchVerif expects that a correct patch should not
  • Increase mission completion time (Timeliness)
  • Increase battery consumption (Efficiency)
  • Increase position errors (Precise navigation)
  • Increase instability (Stability)
  • Cause a new error states (State consistency)
Analyzing Patch Type

\[ I_1, I_2, I_3 = \{ P_1, P_2, P_3, P_4, P_5 \} \]

1. Patch is related to ground speed?
   - Yes
   - No
     - \[ I_1 = \{ P_3, P_4, P_5 \} \]

2. Patch is related to an environmental factor?
   - Yes
     - \[ I_2 = \{ P_5 \} \]
   - No
     - \[ I_2 = \{ P_5 \} \]

3. Patch is adding a new feature?
   - Yes
     - \[ I_3 = \{ P_3, P_5 \} \]
   - No
     - \[ I_3 = \{ P_3, P_5 \} \]

Invariant = \( I_1 \cap I_2 \cap I_3 \)
5 Bug Oracle

• **Solution**: Employ support vector machines (SVMs) to infer whether a patch is faulty or correct

### Table:

<table>
<thead>
<tr>
<th></th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
<th>P₅</th>
</tr>
</thead>
<tbody>
<tr>
<td>RV ᵇ unpatched</td>
<td>51</td>
<td>4530</td>
<td>2.15</td>
<td>3.9</td>
<td>(\text{gyro}=\text{OK}, \text{gps}=\text{OK}) (\text{flight stage}={\text{takeoff, flying, land}, \ldots})</td>
</tr>
<tr>
<td>RV ᵇ patched</td>
<td>600</td>
<td>71154</td>
<td>3.04</td>
<td>5.6</td>
<td>(\text{gyro}=\text{OK}, \text{gps}=\text{OK}) (\text{flight stage}={\text{takeoff, flying, crash}, \ldots})</td>
</tr>
<tr>
<td>Difference</td>
<td>549</td>
<td>66624</td>
<td>0.89</td>
<td>1.7</td>
<td>(\text{flight stage}={\text{land, crash}})</td>
</tr>
</tbody>
</table>

P₁: Timeliness  
P₂: Efficiency  
P₃: Precise navigation  
P₄: Stability  
P₅: State consistency
Evaluation Results

• RV control software
  • ArduPilot, PX4

• Dataset
  • 80 already known correct patches
  • 80 already known faulty patches

• Results
  • PatchVerif achieved, on average, 94.9% F1-score
### Analysis of the Discovered Bugs

<table>
<thead>
<tr>
<th></th>
<th>Unstable attitude/position control</th>
<th>Fail to finish a mission</th>
<th>Crash into ground</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total (115)</td>
<td>36 (31.3%)</td>
<td>2 (1.7%)</td>
<td>77 (67%)</td>
</tr>
</tbody>
</table>
While PatchVerif classifies patches as faulty, they are actually correct patches.

2 false positives
- Patched version shows increased position errors compared to unpatched version. Yet, they are developers’ intension.
  - e.g., sailboat and spline & straight waypoints
False Negatives

• While PatchVerif classifies patches as correct, they are actually faulty patches

• 6 false negatives
  • Why? The 6 faulty patches do not impact the RV’s physical behaviors
    • e.g., Display messages, logging, and camera
Case Study (Object Avoidance)

• The RV’s object avoidance
  • Dijkstra’s path planning algorithm
    • Create safe areas around any object or geo-fenced location
    • Find the shortest path
  • “simple avoidance” algorithm
    • Stop the RV or go backward if the RV enters a safety margin area
Case Study (Object Avoidance Failure)

- Dijkstra’s path planning algorithm makes the RV enter the safe area.

```
  \[\text{Safe area calculated from the geo-fence}\]
```

```
  \[\text{\textquote{simple avoidance} algorithm causes the RV to move backward because the RV also enters the safety margin area.}\]
```

Result: Repeatedly move back and forth near the board of a margin area, and it is unable to complete its mission.