# 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

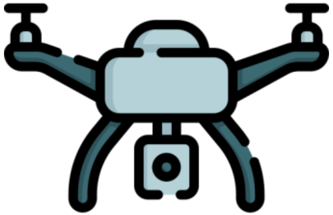
PURDUE
UNIVERSITY

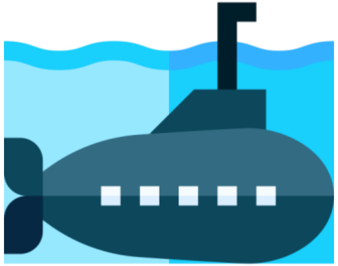
CERIAS

PurSecLab

# 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

1) H.S Kim et al., “PGPATCH: Policy-Guided Logic Bug Patching for Robotic Vehicles”, S&P 2022.

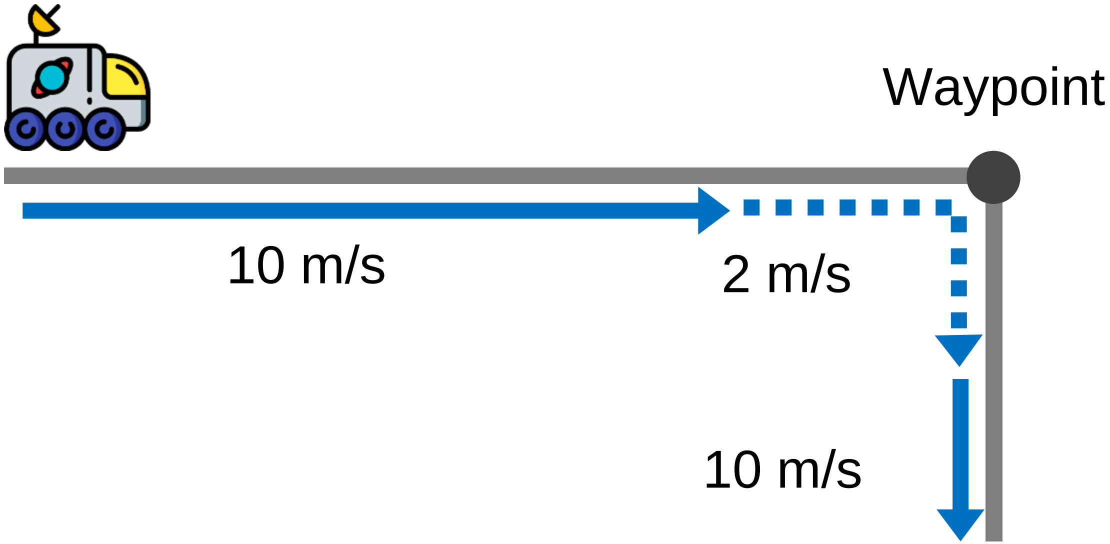
Q: Why is creating patches for RV control software challenging?

**A: Tracking patch-introduced behavioral modifications is difficult.**

# Pivot Turn (1)



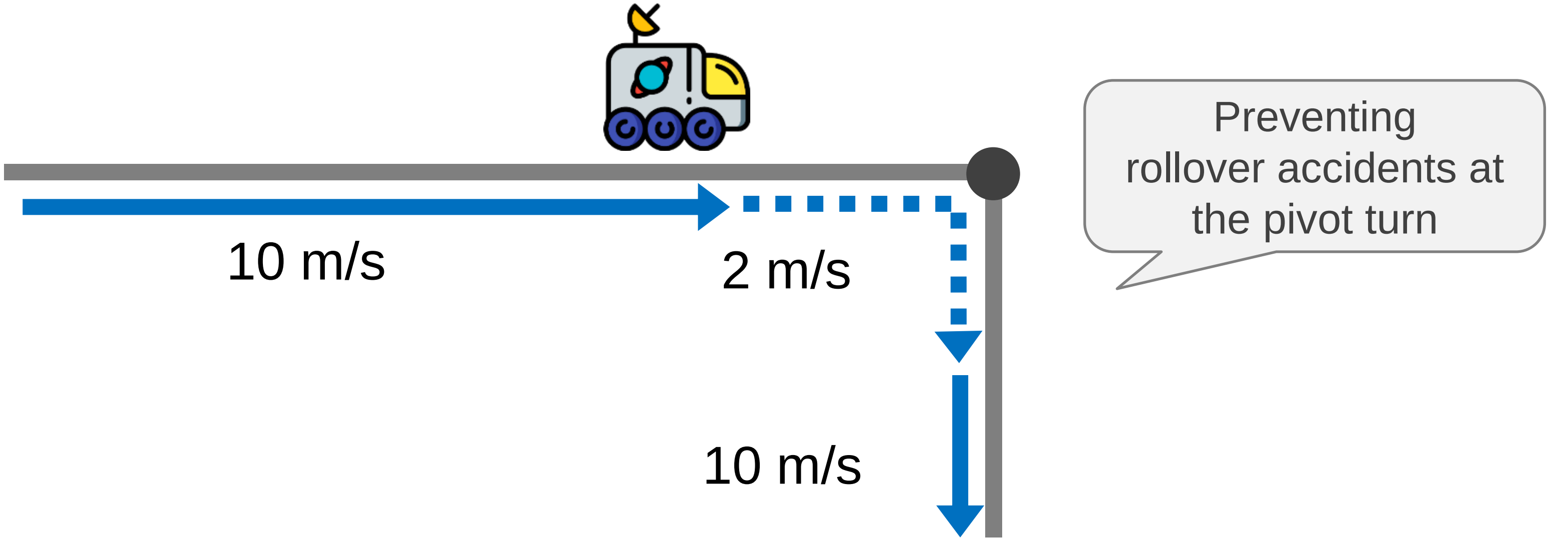
- When a rover is near a corner
  - The vehicle should reduce its speed, turn towards the next waypoint, and continue the navigation.



# 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



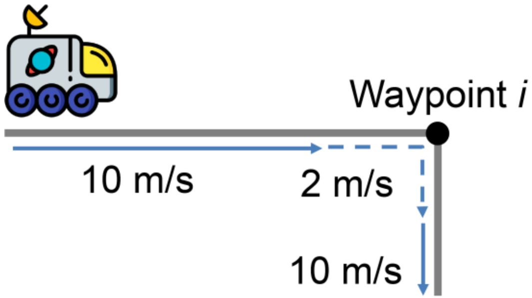
```
void Mode::navigate_to_waypoint() {
- float desired_speed = g2.wp_nav.get_speed();
+ float desired_speed = g2.wp_nav.get_desired_speed();
}
```

Returns slower speed while the RV gets near to a waypoint

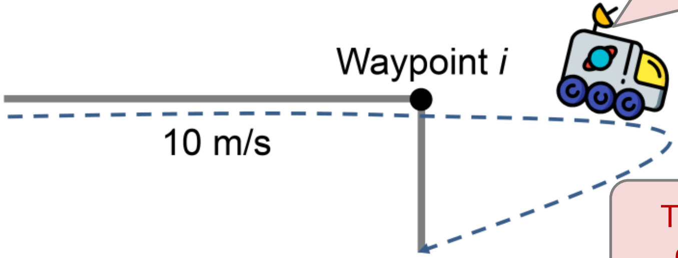
Returns a constant speed set by a configuration parameter

<A faulty patch in a RV control software>

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>

This RV can roll overed due to its high speed.

PURDUE UNIVERSITY CERIAS PurSecLab

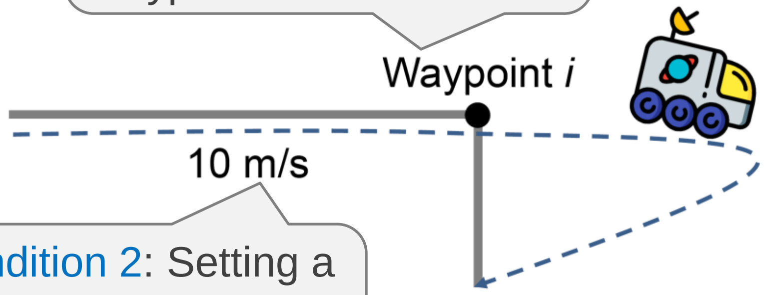
Why do test cases created by developers fail to detect the faulty patch?

# Test Cases Created by Developers



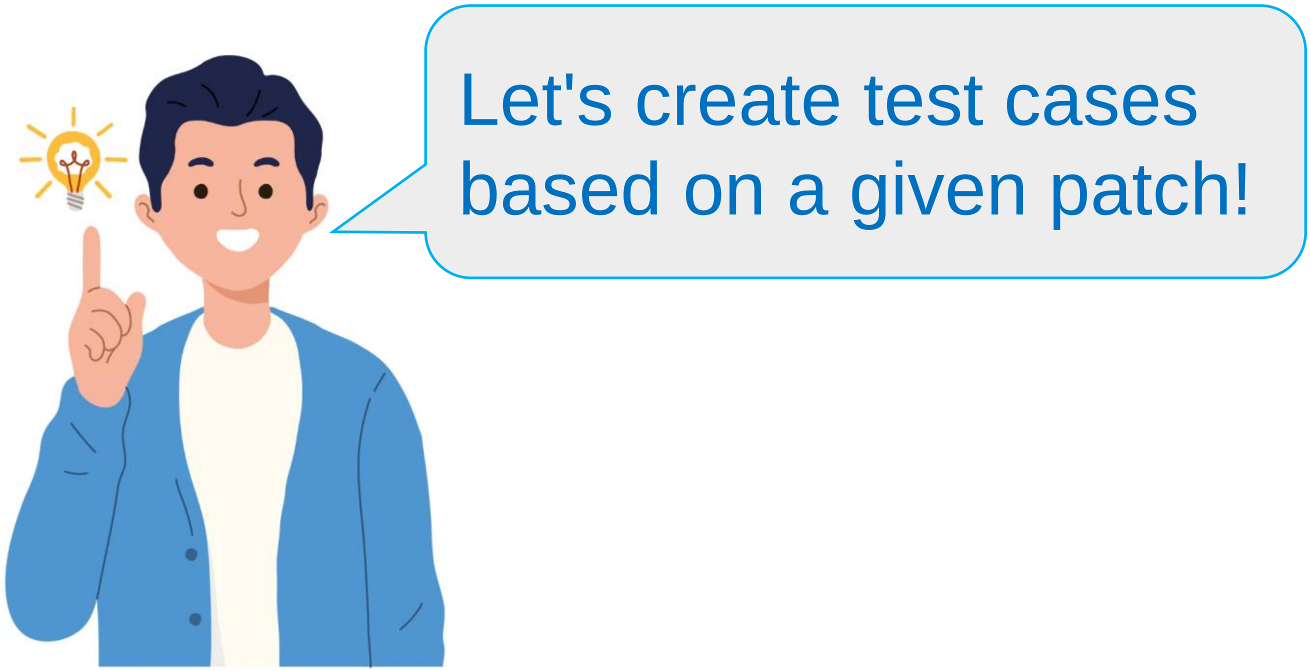
- Manually created test cases do not exercise the physical conditions that trigger the buggy behavior.

Condition 1: Creating a sharp corner through waypoints

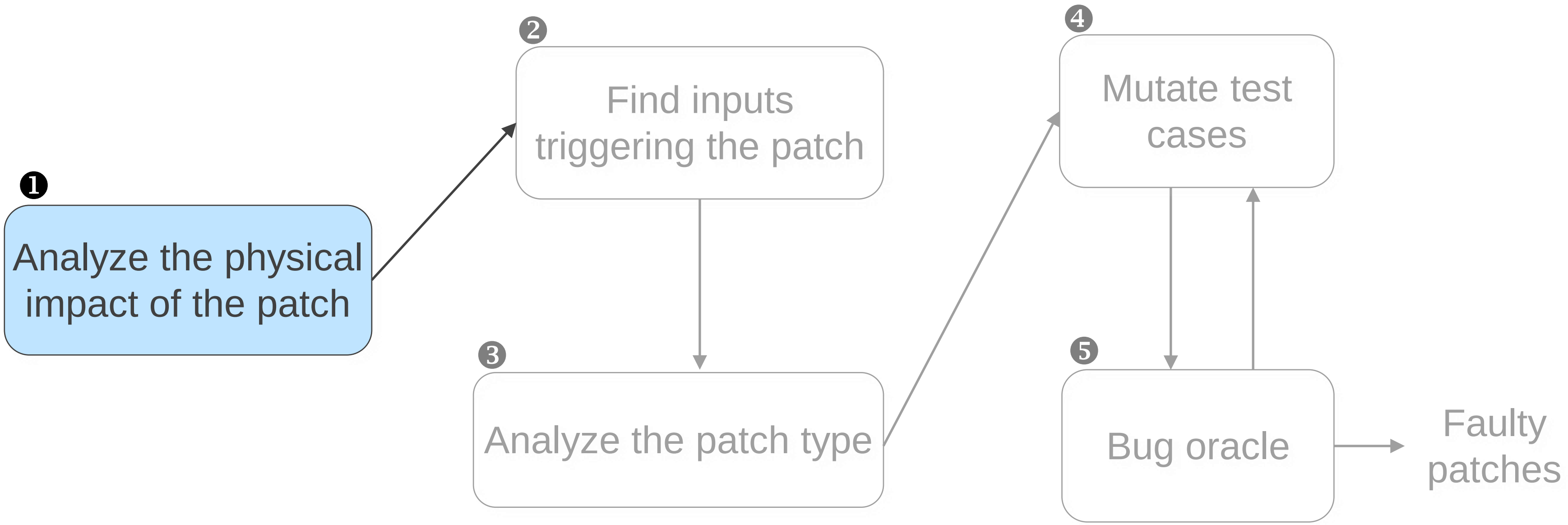



Condition 2: Setting a high ground speed (e.g., 10 m/s)

# Main Idea of PatchVerif



# Overview of PatchVerif



# ❶ Analyze Physical Impact of Patches (1/3)

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

```
+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)) { ... }
```

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

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

# ❶ 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

center.get_alt_frame
location&
above_terrain
circle_center
…

A list of terms extracted from a patch

get_alt_frame
location
above_terrain
circle_center
…

After filtering process

**Step 2: Filter out all but nouns from the variable/function names**

# ❶ 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

We call these identified states and environments $\text{Physical}_{\text{set}}$

*get_alt_frame* is related to the *altitude* state

get_alt_frame
location
above_terrain
circle_center
…

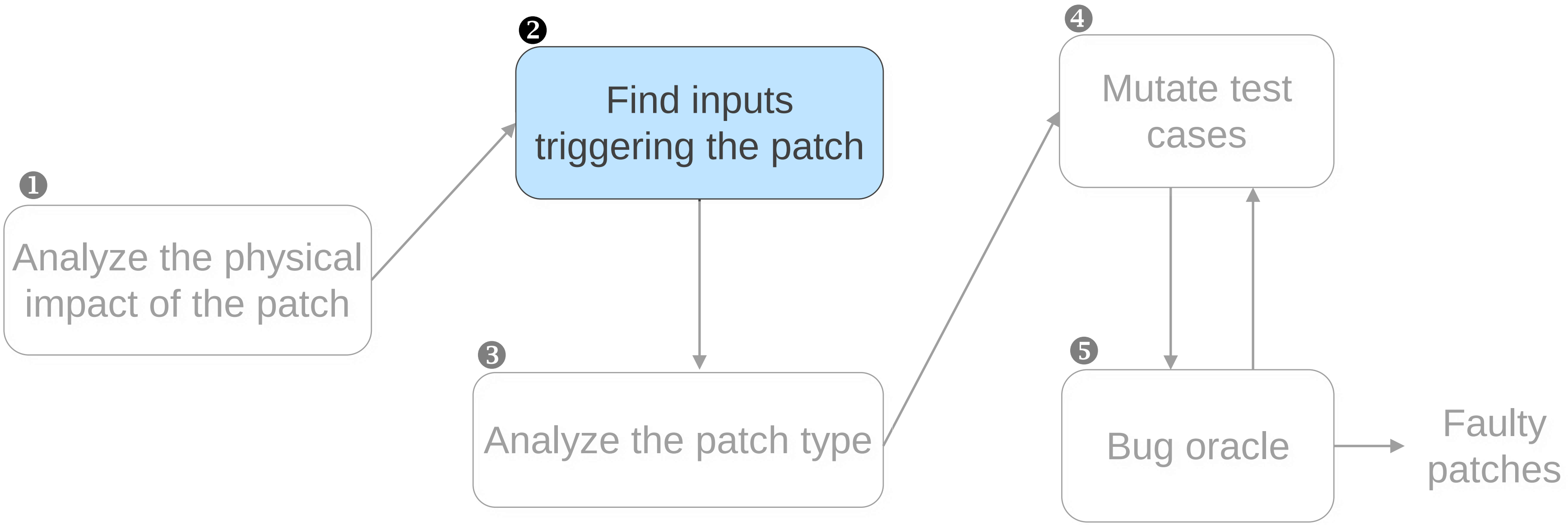
A list of terms extracted from a patch

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

Synonym table

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

# Overview of PatchVerif



# ❷ Find Inputs Triggering Patches

- Goal: Finding inputs (user commands/configuration parameters) triggering the patch code snippet
  - Executing inputs related to the identified $\text{Physical}_{\text{set}}$

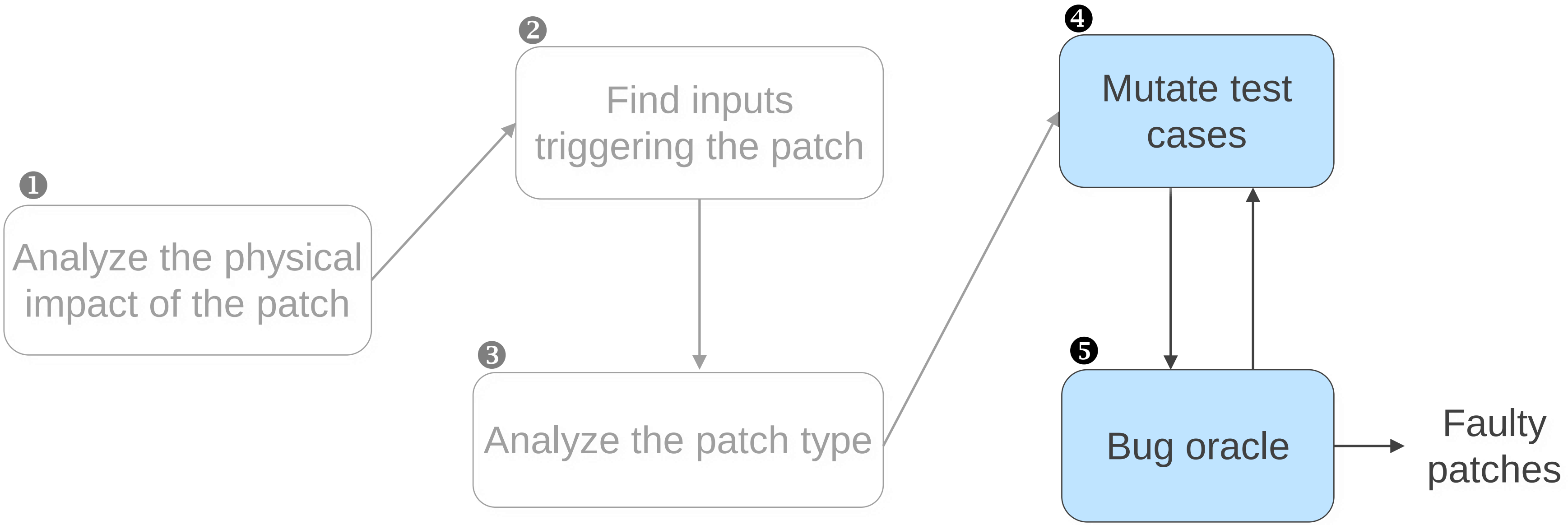
**$\text{Physical}_{\text{set}}$:** location, altitude, flight mode, terrain

CIRCLE flight mode triggers the patch code snippet.

```
+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)) { ... }
```

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

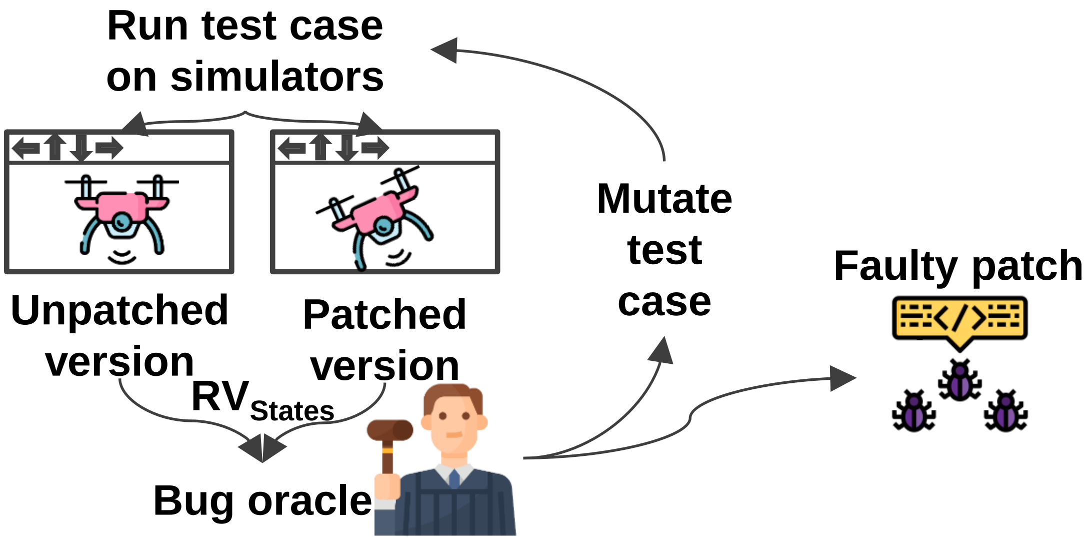
# Overview of PatchVerif



# ❹ Mutate Test Cases



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

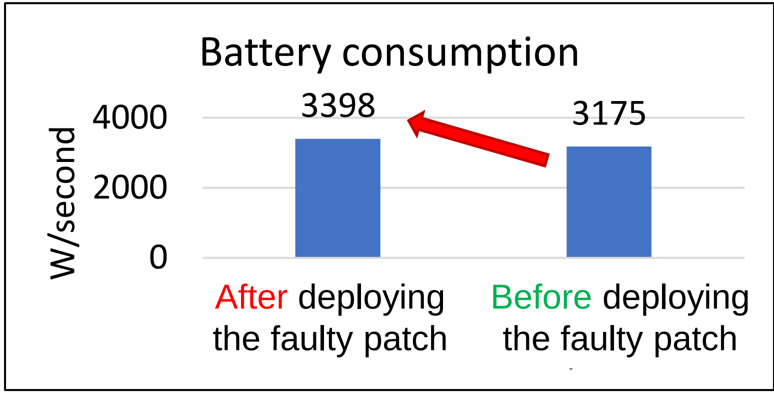


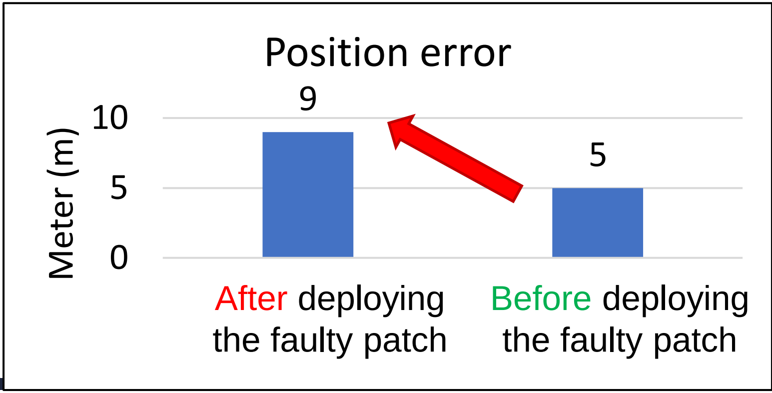
# ❹ Mutate Test Cases

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

<**After** deploying the faulty patch>

<**Before** deploying the faulty patch>

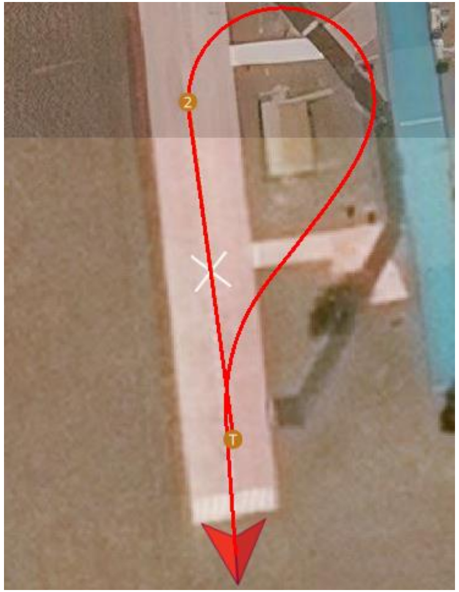


# ❺ Bug Oracle

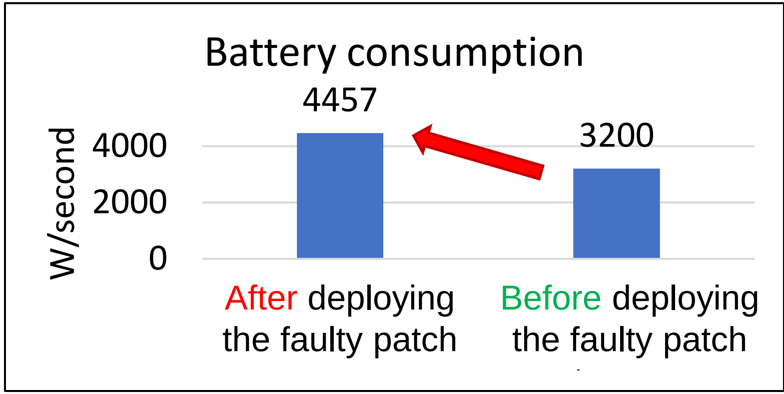


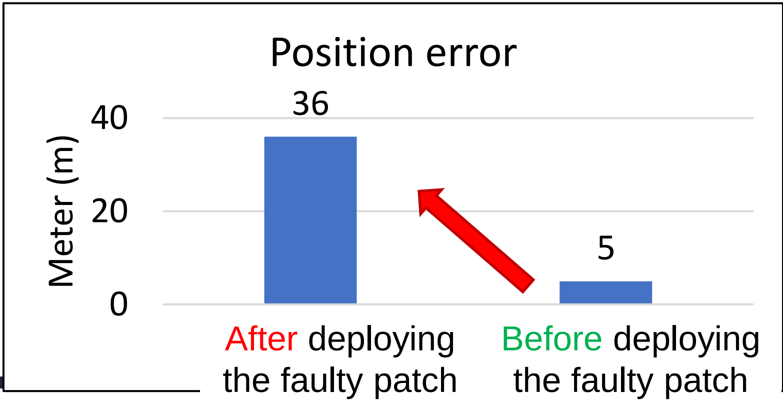
- Mutating the identified inputs to test the patch
  - Increasing the rover's speed (**10 m/s**)



<**After** deploying the faulty patch>

<**Before** deploying the faulty patch>




PURDUE UNIVERSITY CERIAS PurSecLab

# 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

Demo video: https://youtu.be/TWK5lFPlLB4

# 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**

DARPA PURDUE UNIVERSITY CERIAS PurSecLab Office of Naval Research ONR Science & Technology

# 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

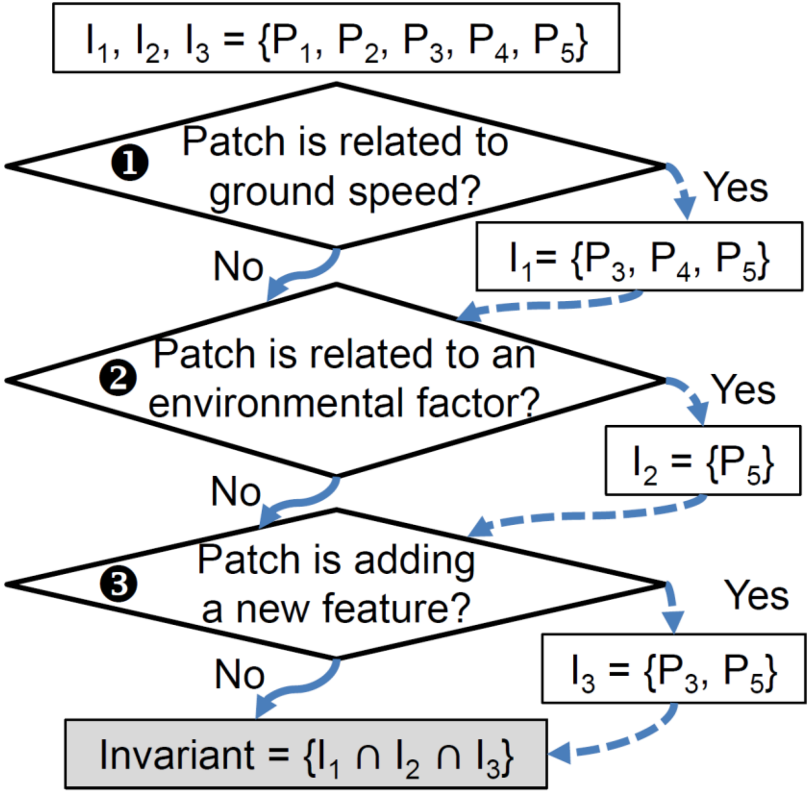
Q: Why do we use a name-based matching rather than taint analysis?

**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)

# ❸ Analyzing Patch Type

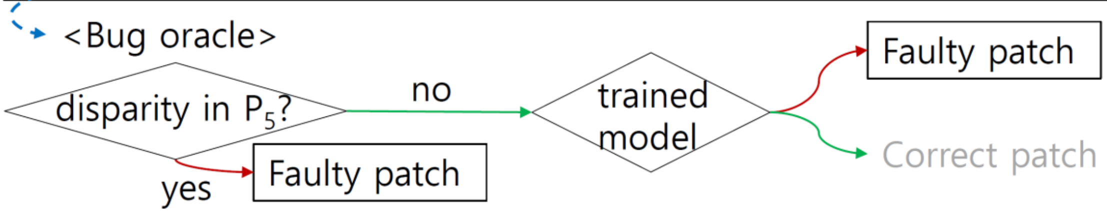



$P_1$: Timeliness
$P_2$: Efficiency
$P_3$: Precise navigation
$P_4$: Stability
$P_5$: State consistency

# ❺ Bug Oracle

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

| | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| $RV_{unpatched}$ | 51 | 4530 | 2.15 | 3.9 | *gyro*={OK}, *gps*={OK}<br>*flight stage*={takeoff, flying, land}, … |
| $RV_{patched}$ | 600 | 71154 | 3.04 | 5.6 | *gyro*={OK}, *gps*={OK}<br>*flight stage*={takeoff, flying, crash}, … |
| Difference | 549 | 66624 | 0.89 | 1.7 | *flight stage*={land, crash} |




$P_1$: Timeliness
$P_2$: Efficiency
$P_3$: Precise navigation
$P_4$: Stability
$P_5$: 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

| | Unstable attitude/position control | Fail to finish a mission | Crash into ground |
|---|---|---|---|
| Total (115) | 36 (31.3%) | 2 (1.7%) | 77 (67%) |

# False Positives

- 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

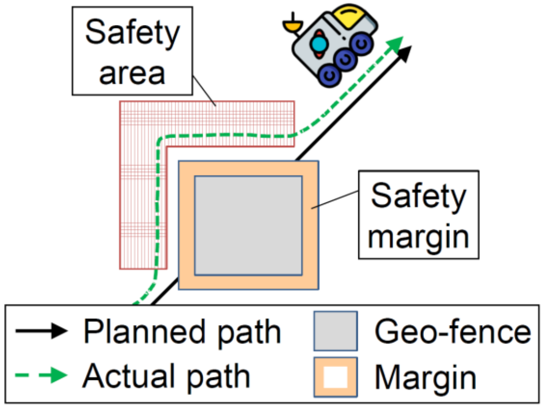
# False Negatives

- 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

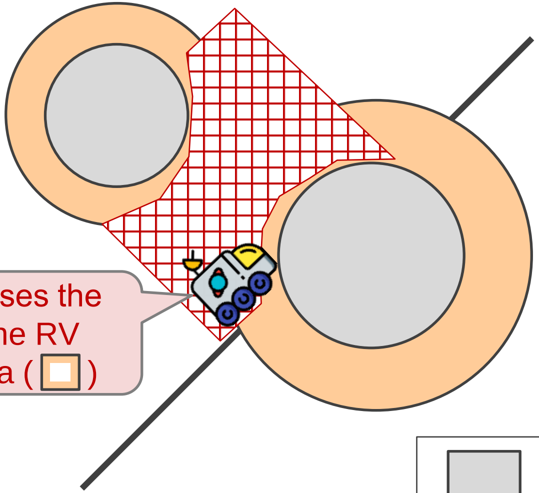
# Case Study (Object Avoidance)

- 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 ( )



“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**

Geo-fence

Safety margin

Safe area calculated from the geo-fence