Defeating Logic bugs in Robotic Vehicles

Hyungsub Kim
Purdue University

CS 490 Software Security
November 16, 2023

About Me1)

- A PhD candidate in Purdue CS
  - Joined in 2018
  - Working on how to apply static and dynamic analysis to robotic vehicle security
  - Published papers into security conferences (S&P, USENIX Security, NDSS)

Details of research topics:

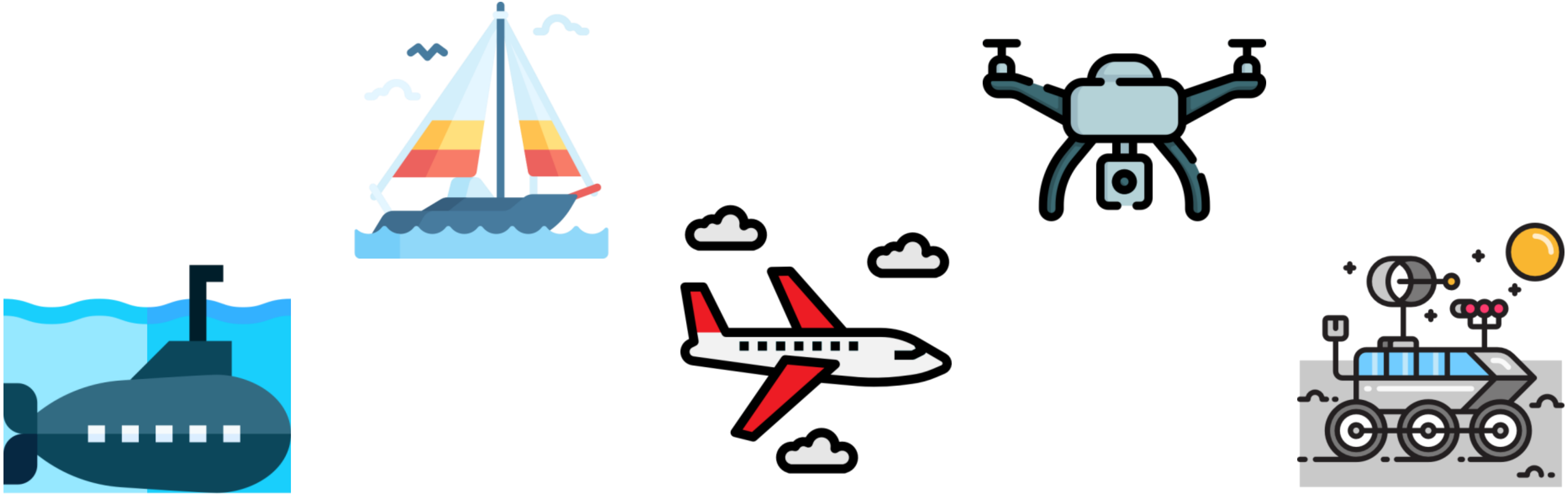
1) Find bugs (fuzzing)
2) Automatically patch the bugs
3) Verify the fixed bugs

---

1) https://kimhyungsub.github.io

What are Robotic Vehicles (RVs)?

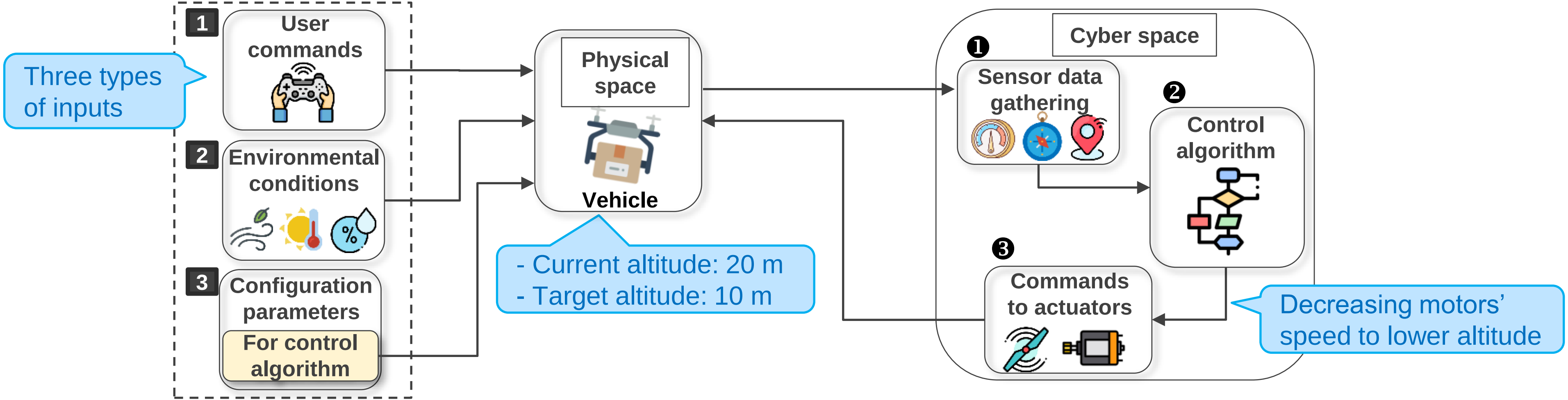
- Vehicles that move *autonomously* on the ground, in the air, on the sea, under the sea, or in space



Workflow of Robotic Vehicles (RVs)

- Physical space
  - Attitude, altitude, speed, etc.
- Cyber space
  - Measuring the RV's current states
  - Adjusting actuators to reach target physical states

Challenge 1: High-dimensional input spaces of RVs

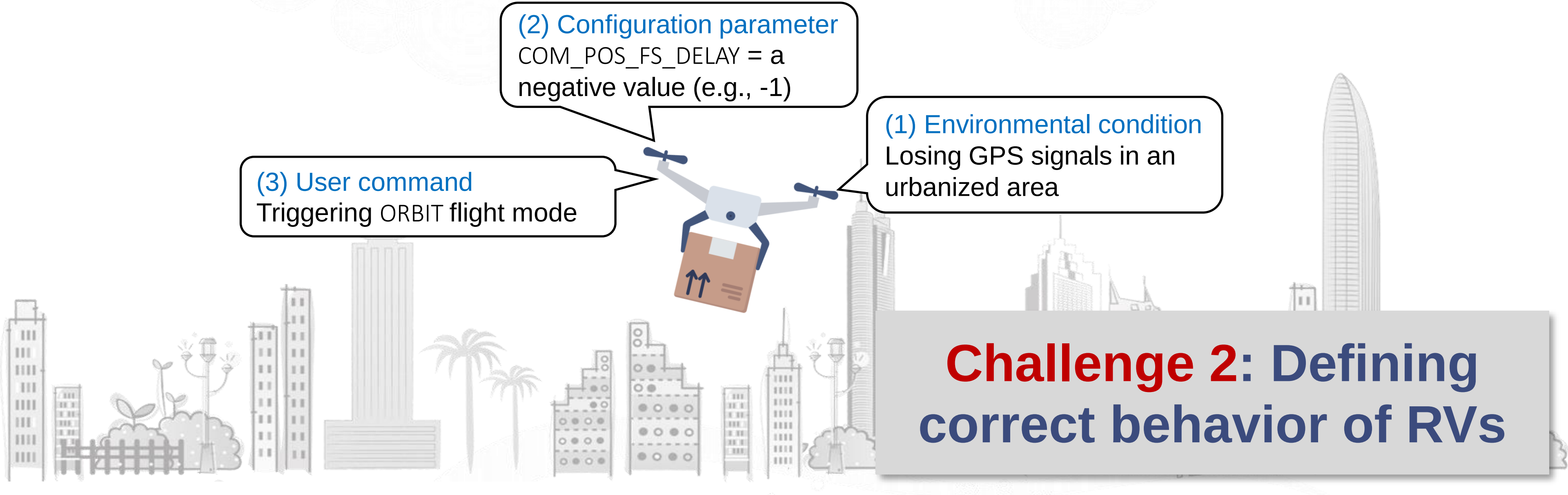


What are Logic Bugs?

1) A policy described in PX4 1) documentation:

- “When time exceeds COM_POS_FS_DELAY seconds after GPS loss is detected, the GPS fail-safe must be triggered”

2) Yet, PX4 fails to trigger the GPS fail-safe under specific physical conditions.



(2) Configuration parameter
COM_POS_FS_DELAY = a negative value (e.g., -1)

(1) Environmental condition
Losing GPS signals in an urbanized area

(3) User command
Triggering ORBIT flight mode

Challenge 2: Defining correct behavior of RVs

---

1) PX4: A popular open-source RV control software

How Common are Logic Bugs?

Analysis of existing bugs in popular RV control software

- 98.2% of them (1,234/1,257) are **logic bugs.**
- 1.8% of them (23/1,257) are memory corruption bugs.

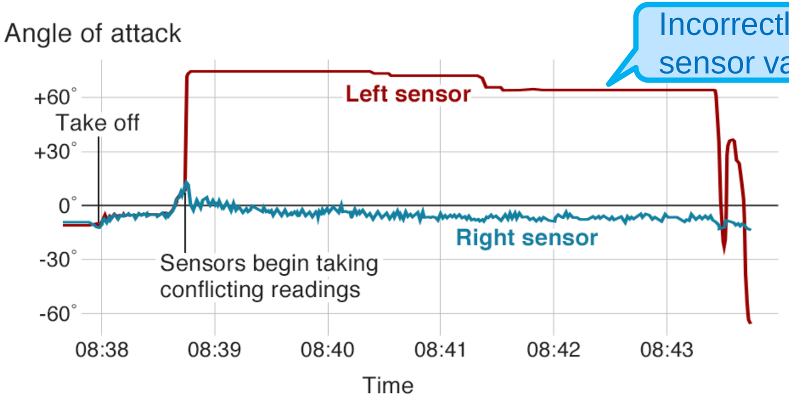
- Attackers can exploit logic bugs.

Logic Bug in Real World

- Boeing-737 Max airplanes
  - Crashed due to a design flaw
  - Lowered its altitude based on only one broken sensor

How can we find such a critical bug in flight control software? Um… fuzzing?

The plane's sensors took different readings



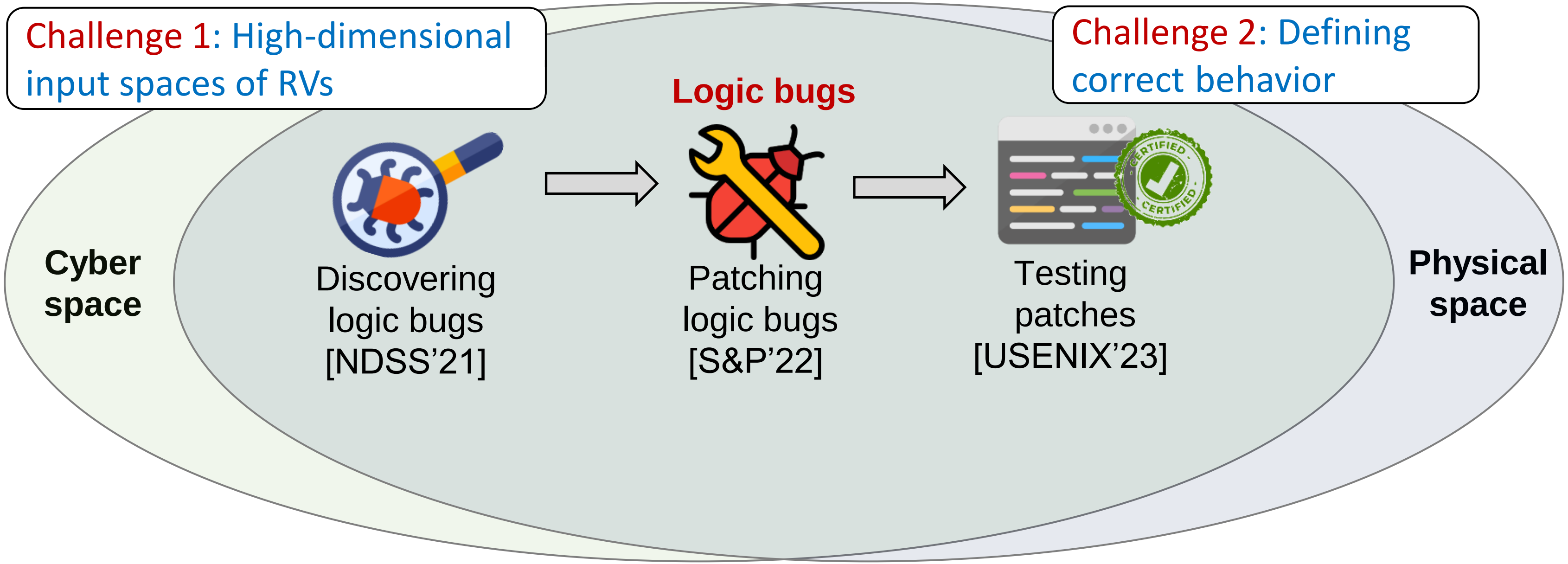
Incorrectly measured sensor values

Source: Ethiopian Aircraft Accident Investigation Bureau

https://www.dailymail.co.uk/news/article-7056177/US-investigators-believe-bird-strike-factor-Ethiopian-Airlines-Boeing-737-Max-8-crash.html

Overview of My Research

- Interplay between cyber and physical space creates challenges.



Challenge 1: High-dimensional input spaces of RVs

Challenge 2: Defining correct behavior

Logic bugs

Cyber space

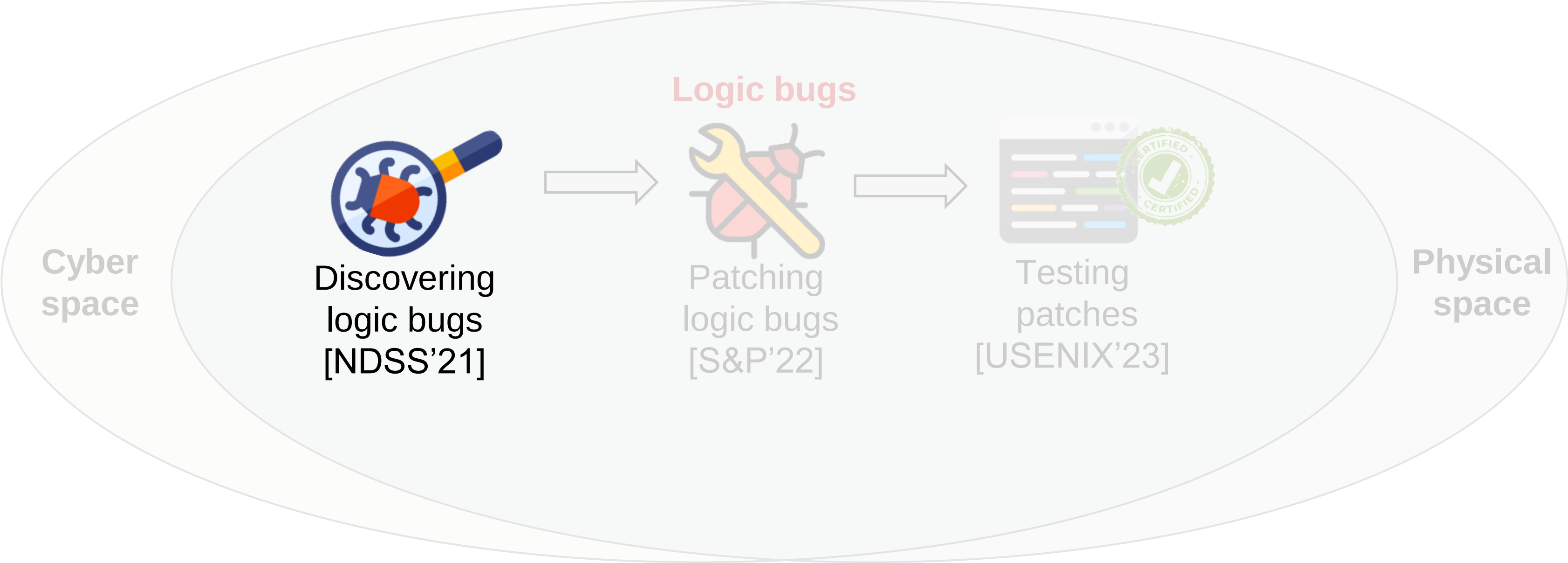
Discovering logic bugs [NDSS’21]

Patching logic bugs [S&P’22]

Testing patches [USENIX’23]

Physical space

Discovering Logic Bugs in RVs



Logic bugs

Cyber space

Discovering logic bugs [NDSS'21]

Patching logic bugs [S&P'22]

Testing patches [USENIX'23]

Physical space

"PGFuzz: Policy-Guided Fuzzing for Robotic Vehicles",
Hyungsub Kim, Muslum Ozgur Ozmen, Antonio Bianchi, Z. Berkay Celik, Dongyan Xu, **NDSS 2021**.

Limitations of Previous Fuzzers

1) Can existing fuzzers discover logic bugs?

What about traditional fuzzers (AFL, libFuzzer)? No

- Mutation: Code coverage
- Bug oracle: Memory access violation

2) To detect logic bugs, we need to tackle two challenges (C)

- (C1) Know the RV’s correct behaviors
- (C2) Reduce high-dimensional input spaces

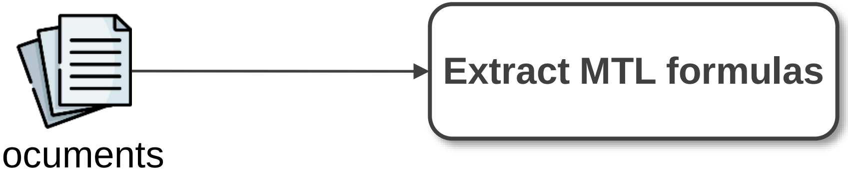
Q: How can we formally define the correct behavior of RVs?

A: Creating MTL[1) formulas from documentation **and** comments **that describe expected behavior of RVs**

---

1) MTL: Metric Temporal Logic

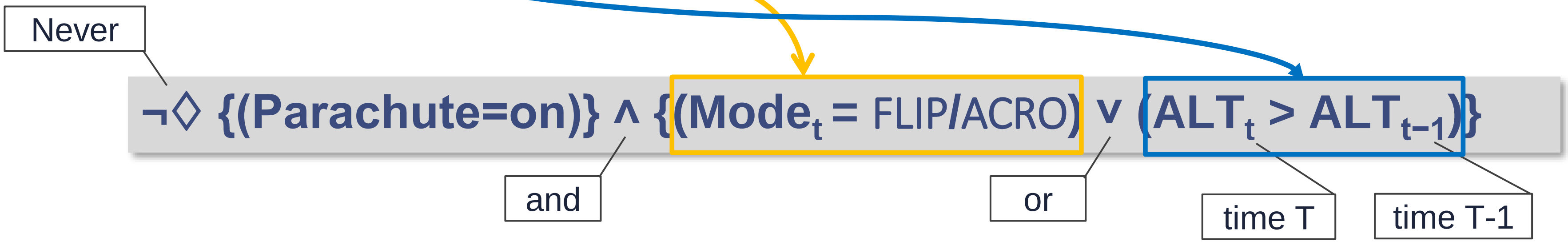
Defining MTL Formulas



Documents

Extract MTL formulas

> *A vehicle must not deploy a parachute when the vehicle is:*
> 1) *In FLIP or ACRO flight modes*
> 2) *Climbing*



$$\neg\Diamond\ \{(\text{Parachute=on})\} \land \{\boxed{(\text{Mode}_t = \text{FLIP/ACRO})} \lor \boxed{(\text{ALT}_t > \text{ALT}_{t-1})}\}$$

Never

and

or

time T

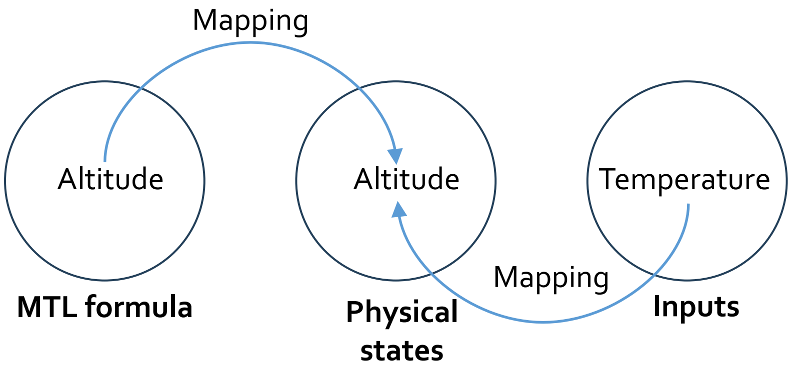
time T-1

The formula is created in the form of Metric temporal logic (MTL).

Q: How can we reduce the high-dimensional input spaces of RVs?

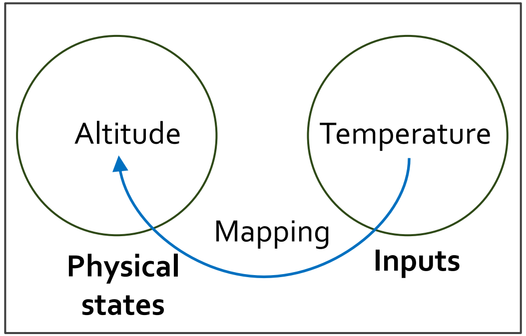
A: Only testing inputs relevant to MTL formulas

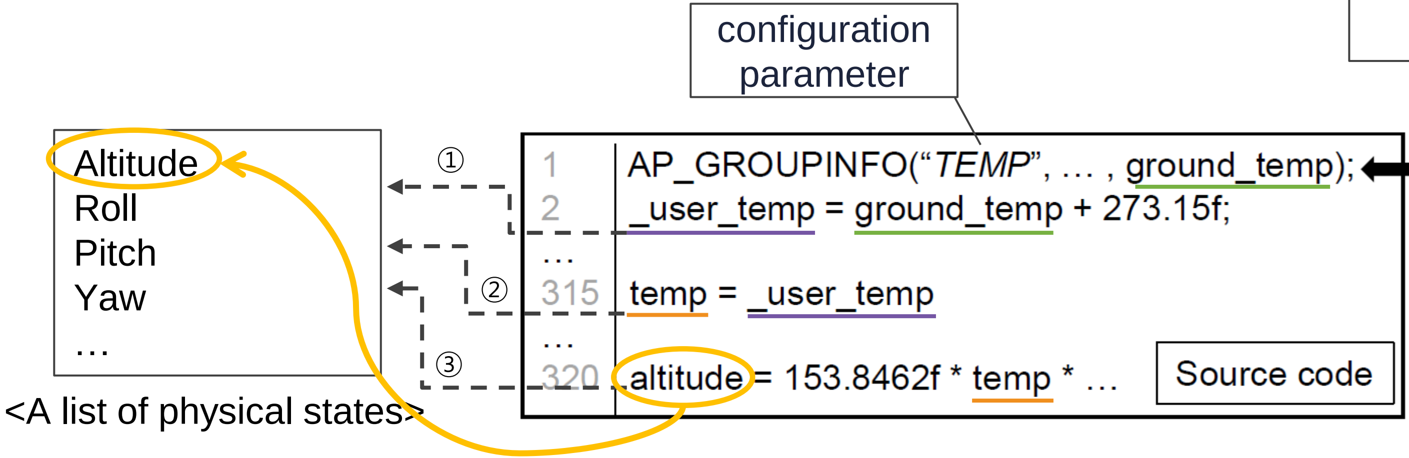
$$\neg\Diamond \{(\text{Parachute}=\text{on})\} \wedge \{(\text{Mode}_t = \text{FLIP/ACRO}) \vee (\text{ALT}_t > \text{ALT}_{t-1})\}$$



Mapping Config. Parameters to Physical States

- Static analysis to identify which physical states are affected by each configuration parameter





Altitude
Roll
Pitch
Yaw
…

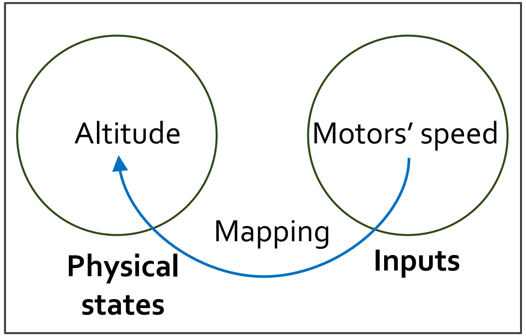
<A list of physical states>

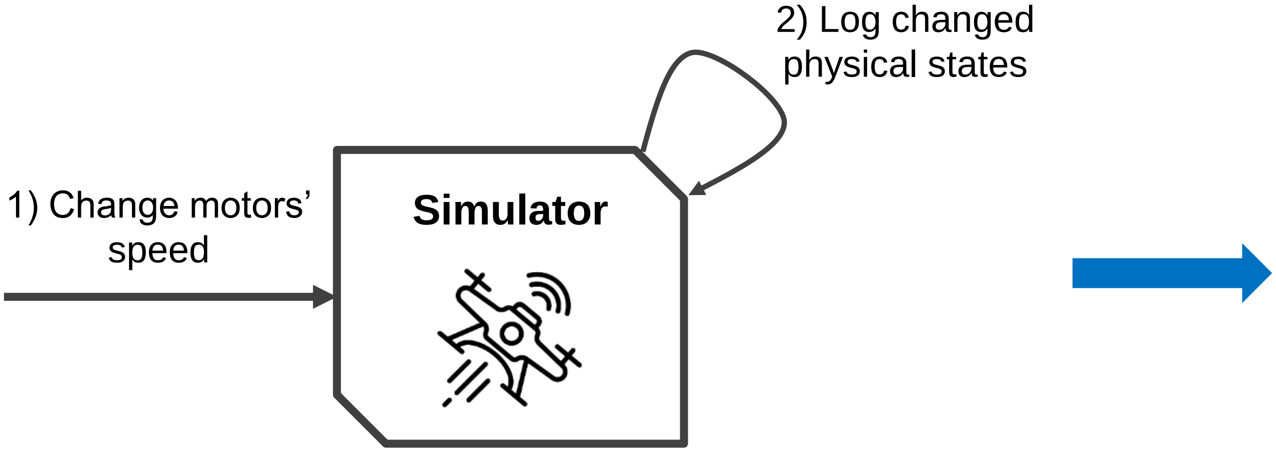
```
AP_GROUPINFO("TEMP", … , ground_temp);
_user_temp = ground_temp + 273.15f;
…
temp = _user_temp
…
altitude = 153.8462f * temp * …
```

Starting point for a def-use chain of *TEMP* parameter

Mapping Other Types of Inputs to Physical States

- How to map environmental conditions and user commands to each term from source code? Use an RV simulator!



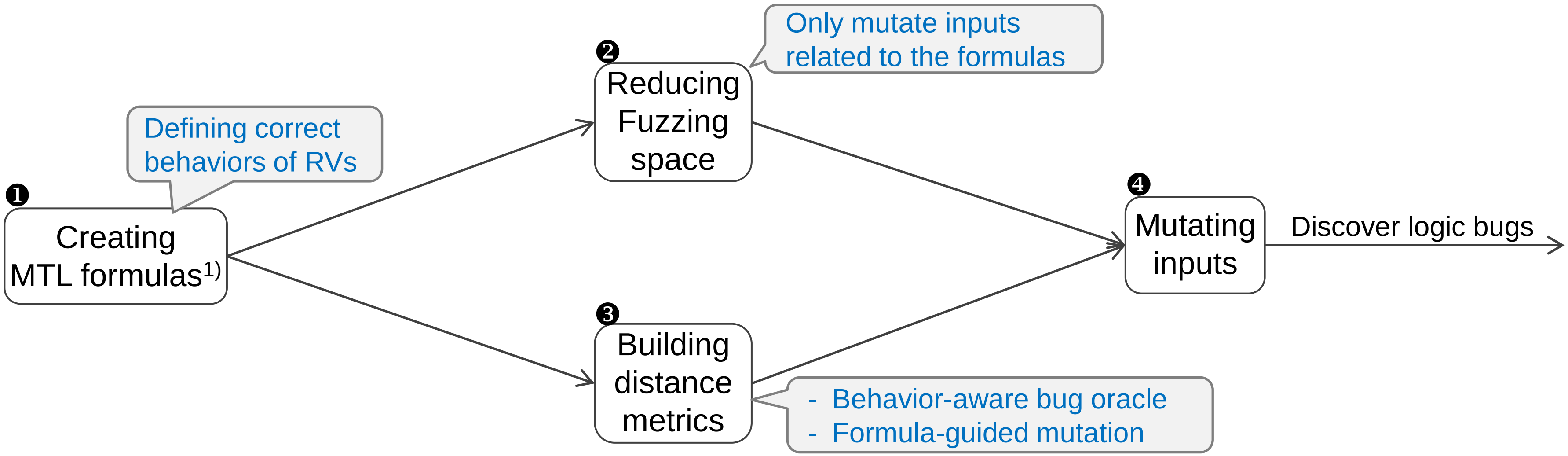


- Heading
- Throttle
- **Altitude**
- Climb

<Changed physical states according to motors' speed>

PGFuzz: Temporal Logic-Guided Fuzzing for RVs

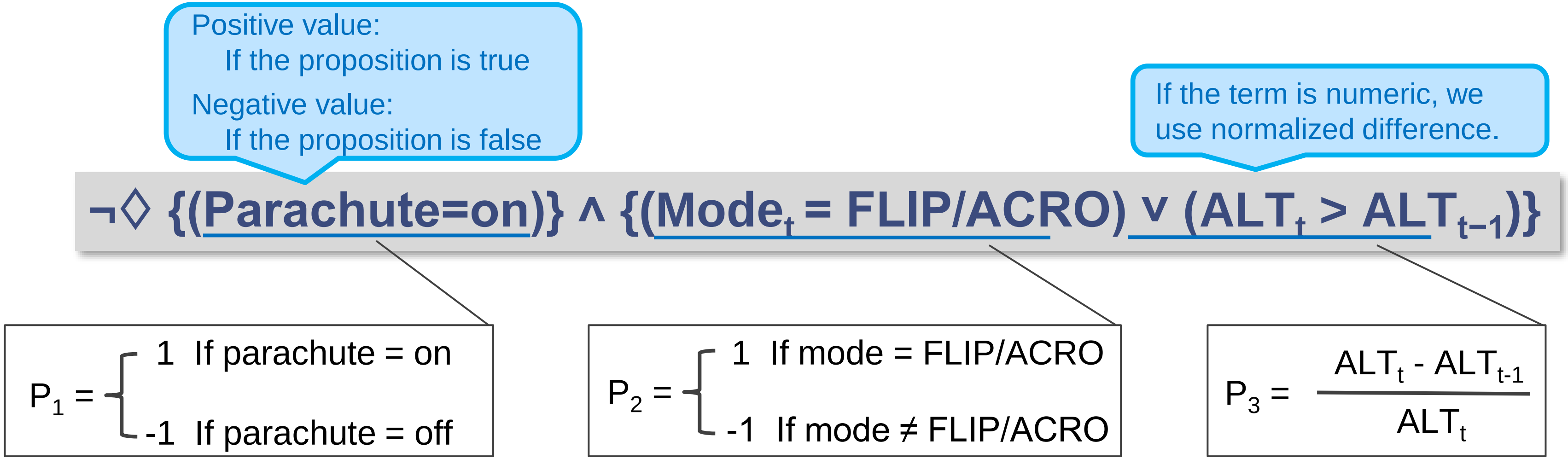
- **Logic bug-finding tool**



1) MTL: Metric Temporal Logic

Two types of distances to mutate inputs

- Propositional distance
  - Goal: efficiently mutating inputs
  - Quantifies how close a proposition to the policy violation



Positive value:
If the proposition is true
Negative value:
If the proposition is false

If the term is numeric, we use normalized difference.

$$\neg\Diamond \{(\mathrm{Parachute{=}on})\} \wedge \{(\mathrm{Mode}_t = \mathrm{FLIP/ACRO}) \vee (\mathrm{ALT}_t > \mathrm{ALT}_{t-1})\}$$

$$P_1 = \begin{cases} 1 & \text{If parachute = on} \\ -1 & \text{If parachute = off} \end{cases}$$

$$P_2 = \begin{cases} 1 & \text{If mode = FLIP/ACRO} \\ -1 & \text{If mode} \neq \text{FLIP/ACRO} \end{cases}$$

$$P_3 = \frac{\mathrm{ALT}_t - \mathrm{ALT}_{t-1}}{\mathrm{ALT}_t}$$

Two types of distances to mutate inputs

- Global distance
  - Goal: detecting a policy violation

$$-1 \times [\mathrm{Min}\{P_1, \mathrm{Max}(P_2, P_3)\}] \begin{cases} \text{Positive value} & \text{if there is no policy violation} \\ \text{Negative value} & \text{if the RV violates the policy} \end{cases}$$

Working example (time T = 1)

$$P_1 = \begin{cases} 1 & \text{If parachute = on} \\ -1 & \text{If parachute = off} \end{cases}$$

$$P_3 = \frac{ALT_t - ALT_{t-1}}{ALT_t}$$

$$P_2 = \begin{cases} 1 & \text{If mode = FLIP/ACRO} \\ -1 & \text{If mode} \neq \text{FLIP/ACRO} \end{cases}$$

$$\boxed{-1 \times [\mathrm{Min}\{P_1, \mathrm{Max}(P_2, P_3)\}]}$$

Randomly select an input and assign a random value to the selected input

Time (T)	Parachute (on/off)	FLIP/ACRO mode (T/F)	Altitude (m)	P_1	P_2	P_3	Global distance	Next input for Time T+1
1	off	false	90	-1	-1	0	1	Motor speed = 1,800[1]
2								
3								
4								

🟩 : RV's current states at time T 🟨 : Calculated distances at time T

---

1) (Motor speed > 1,500) → increasing RV's altitude
(Motor speed < 1,500) → decreasing RV's altitude

Working example (time T = 2)

$$P_1 = \begin{cases} 1 & \text{If parachute = on} \\ -1 & \text{If parachute = off} \end{cases}$$

$$P_3 = \frac{ALT_t - ALT_{t-1}}{ALT_t}$$

$$\boxed{-1 \text{ X } [\text{Min}\{P_1, \text{Max}(P_2, P_3)\}]}$$

$$P_2 = \begin{cases} 1 & \text{If mode = FLIP/ACRO} \\ -1 & \text{If mode} \neq \text{FLIP/ACRO} \end{cases}$$

1) We log (motor, 1,800) because the input increases P3.

2) PGFUZZ selects an input and assign a random value to the selected input

Time (T)	Parachute (on/off)	FLIP/ACRO mode (T/F)	Altitude (m)	P_1	P_2	P_3	Global distance	Next input for Time T+1
1	off	false	90	-1	-1	0	1	Motor speed = 1,800 1)
2	off	false	100	-1	-1	0.1	1	Motor speed = 1,800 1)
3								
4								

3) When the selected input increased a distance before, we reuse the input and value pair (motor, 1,800)

1) (Motor speed > 1,500) → increasing RV's altitude
(Motor speed < 1,500) → decreasing RV's altitude

Working example (time T = 3)

$$P_1 = \begin{cases} 1 & \text{If parachute = on} \\ -1 & \text{If parachute = off} \end{cases}$$

$$P_3 = \frac{ALT_t - ALT_{t-1}}{ALT_t}$$

$$P_2 = \begin{cases} 1 & \text{If mode = FLIP/ACRO} \\ -1 & \text{If mode} \neq \text{FLIP/ACRO} \end{cases}$$

$$\boxed{-1 \text{ X } [\text{Min}\{P_1, \text{Max}(P_2, P_3)\}]}$$

Time (T)	Parachute (on/off)	FLIP/ACRO mode (T/F)	Altitude (m)	P_1	P_2	P_3	Global distance	Next input for Time T+1
1	off	false	90	-1	-1	0	1	Motor speed = 1,800 1)
2	off	false	100	-1	-1	0.1	1	Motor speed = 1,800
3	off	false	110	-1	-1	0.09	1	Parachute = on
4								

PGFUZZ selects an input

---

1) (Motor speed > 1,500) → increasing RV's altitude
(Motor speed < 1,500) → decreasing RV's altitude

Working example (time T = 4)

$$P_1 = \begin{cases} 1 & \text{If parachute = on} \\ -1 & \text{If parachute = off} \end{cases}$$

$$P_3 = \frac{ALT_t - ALT_{t-1}}{ALT_t}$$

$$P_2 = \begin{cases} 1 & \text{If mode = FLIP/ACRO} \\ -1 & \text{If mode} \neq \text{FLIP/ACRO} \end{cases}$$

$$\boxed{-1 \times [\mathrm{Min}\{P_1, \mathrm{Max}(P_2, P_3)\}]}$$

Time (T)	Parachute (on/off)	FLIP/ACRO mode (T/F)	Altitude (m)	P_1	P_2	P_3	Global distance	Next input for Time T+1
1	off	false	90	-1	-1	0	1	Motor speed = 1,800[1]
2	off	false	100	-1	-1	0.1	1	Motor speed = 1,800
3	off	false	110	-1	-1	0.09	1	Parachute = on
4	on	false	112	1	-1	0.02	-0.02	Policy violation!

Vehicle must not increase its altitude

---

1) (Motor speed > 1,500) → increasing RV's altitude
(Motor speed < 1,500) → decreasing RV's altitude

Evaluation

- RV control software
    - ArduPilot, PX4, and Paparazzi

- 56 extracted policies
    - Fuzzing 48 hours per each control software
    - Violating 14 policies in the three-control software

- Found 156 bugs

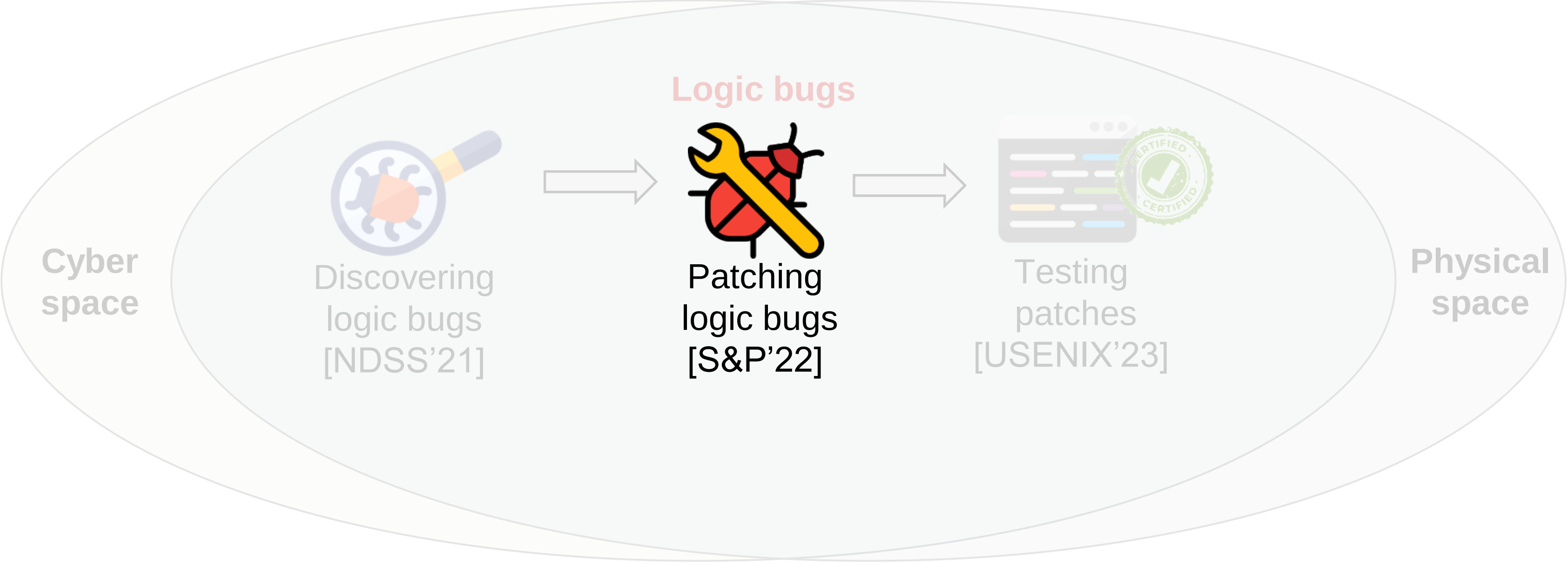
Takeaways from PGFuzz

- A new fuzzing approach to find logic bugs
  - Behavior-aware bug oracle
    - Leverage MTL formulas
  - Customized program analysis
    - Mapping a formula to inputs
  - Discovering subtle logic bugs

Challenge 2: Defining correct behavior

Challenge 1: High-dimensional input spaces of RVs

Fixing Logic Bugs in RVs



Logic bugs

Cyber space

Discovering logic bugs [NDSS’21]

Patching logic bugs [S&P’22]

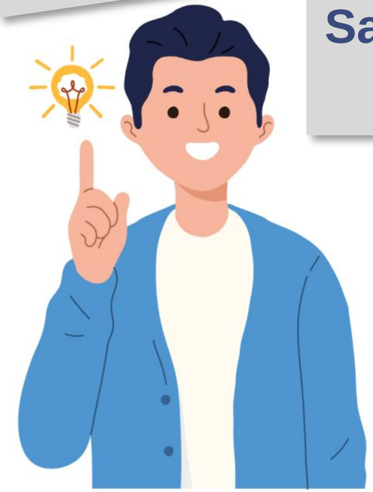
Testing patches [USENIX’23]

Physical space

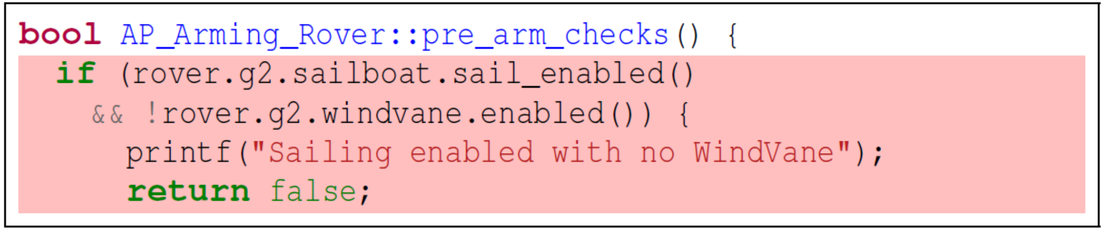
“PGPatch: Policy-Guided Logic Bug Patching for Robotic Vehicles”,
Hyungsub Kim, Muslum Ozgur Ozmen, Z. Berkay Celik, Antonio Bianchi, Dongyan Xu, **S&P 2022**.

Motivation of PGPatch

Can we reuse formulas to fix the found bugs?



Sailboat formula: □ {(armed = false)} ∧ {(SAIL_ENABLE = True) ∧ (WNDVN_TYPE = False) → (pre_arm_checks = error)}



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

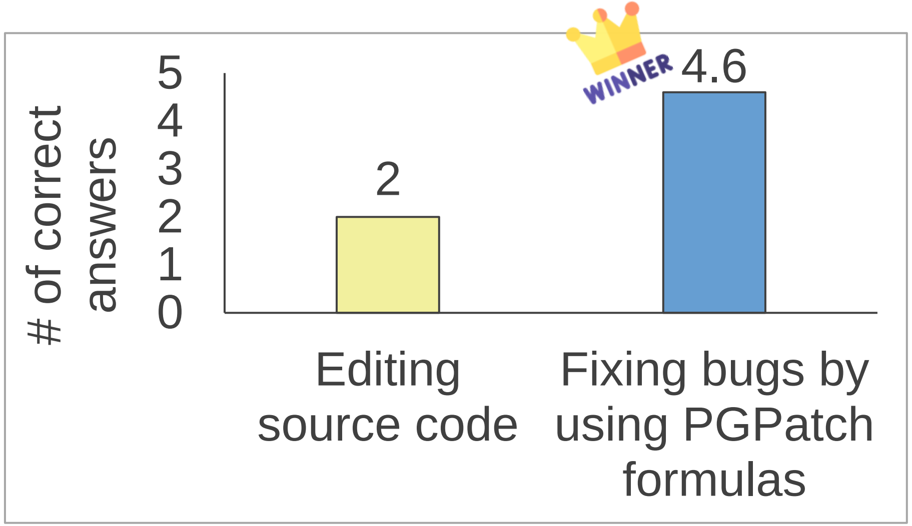
User Study

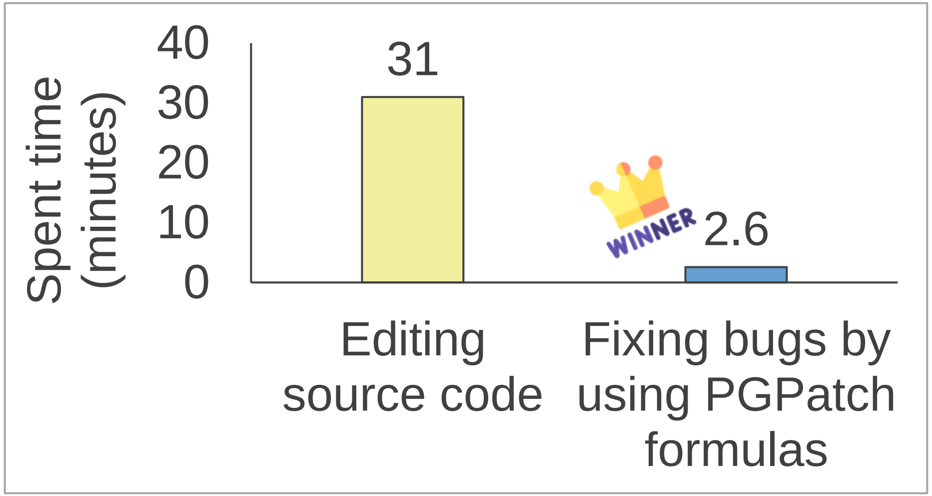
- Aim to determine
  - How efficient PGPatch is in patching logic bugs compared to manual patching

- Method
  - Recruit 6 RV developers and 12 experienced RV users
    - 1 subject was an official ArduPilot developer
  - Ask participants to create:
    - 5 patches using PGPatch
    - 5 corresponding source-level patches

User Study

- Correctness
  - 2 (editing source code) vs. 4.6 (fixing bugs through PGPatch)
- Spent time
  - 31 mins (editing source code) vs. 2.6 mins (fixing bugs through PGPatch)





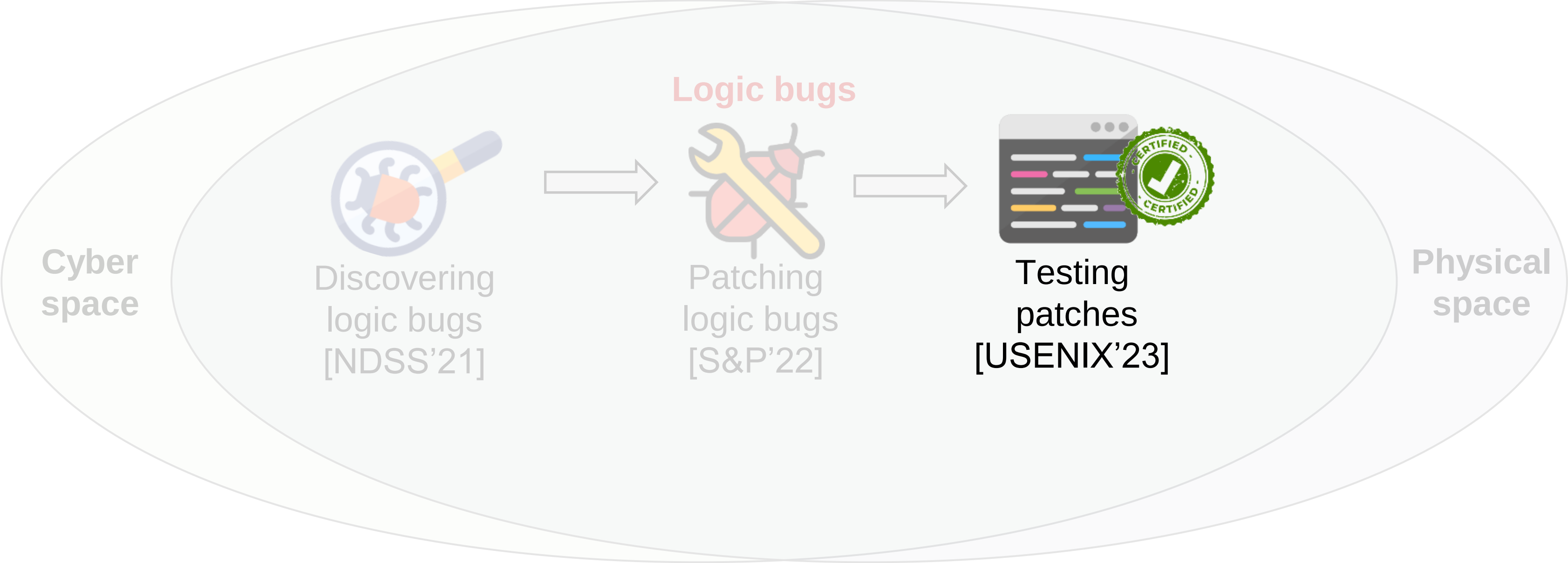
Takeaways from PGPatch

- A new approach to fix logic bugs
  - Reuse existing MTL formulas
  - Less error-prone compared to manually patching bugs
    - Proven by the user study

Challenge 1: High-dimensional input spaces of RVs

Challenge 2: Defining correct behavior

Testing Correctness of Patches



Logic bugs

Cyber space

Discovering logic bugs [NDSS'21]

Patching logic bugs [S&P'22]

Testing patches [USENIX'23]

Physical space

"PatchVerif: Discovering Faulty Patches in Robotic Vehicles",
Hyungsub Kim, Muslum Ozgur Ozmen, Z. Berkay Celik, Antonio Bianchi, Dongyan Xu, **USENIX Security 2023**.

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 prone1)
    - 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

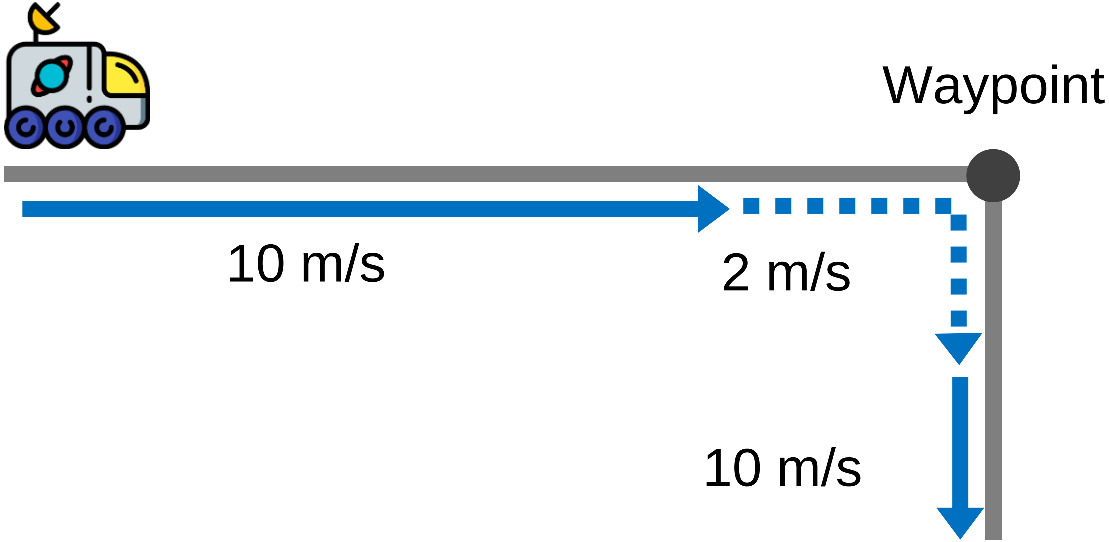
1) Hyungsub Kim et al., “PGPATCH: Policy-Guided Logic Bug Patching for Robotic Vehicles”, S&P 2022.

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

A: Tracking patch-introduced behavioral modifications is difficult.

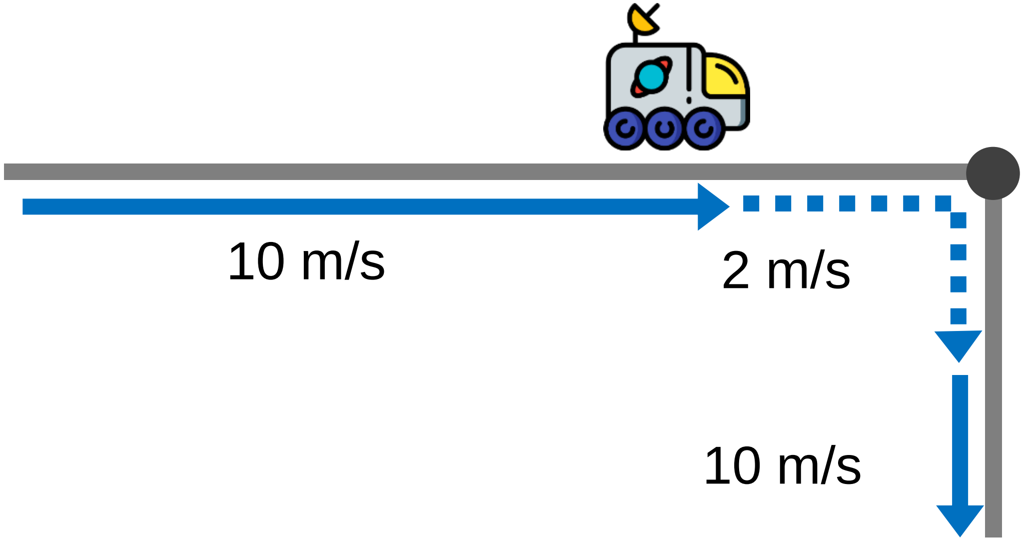
Pivot Turn (1)

- 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.



Preventing rollover accidents at the pivot turn

Motivating Example

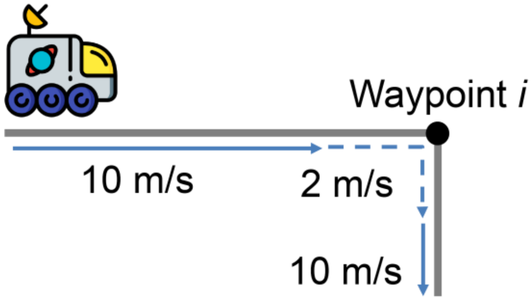
```
void Mode::navigate_to_waypoint() {

- 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

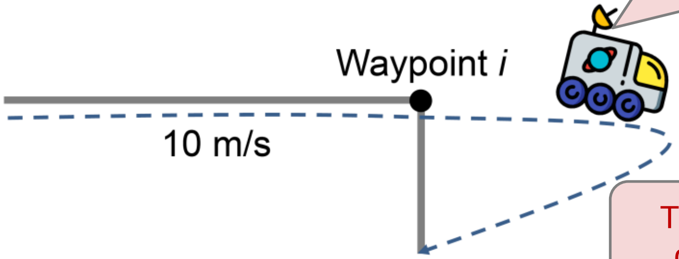
Returns a constant speed set by a configuration parameter

<A faulty patch in a RV control software>

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



<Normal RV behavior before deploying the faulty patch>



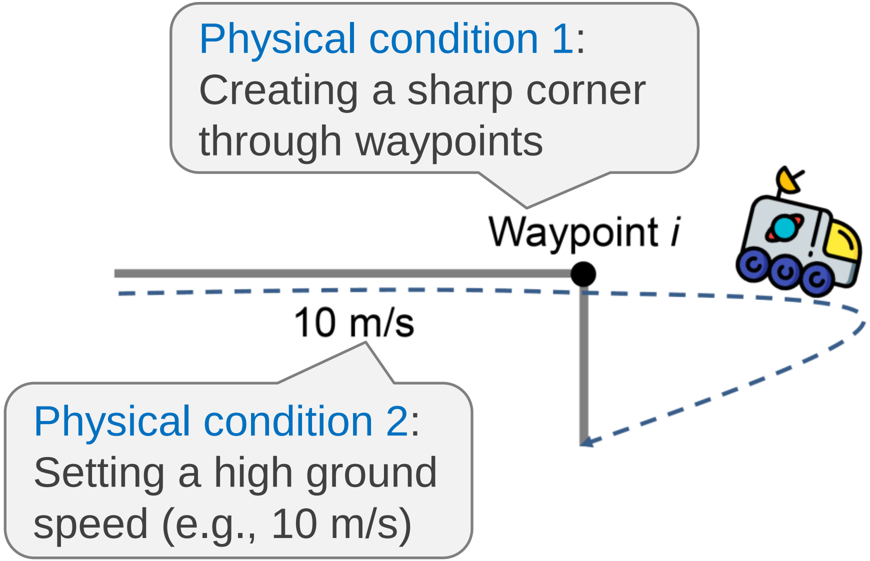
This RV can roll overed due to its high speed.

<Abnormal RV behavior after deploying the faulty patch>

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

Test Cases Created by Developers

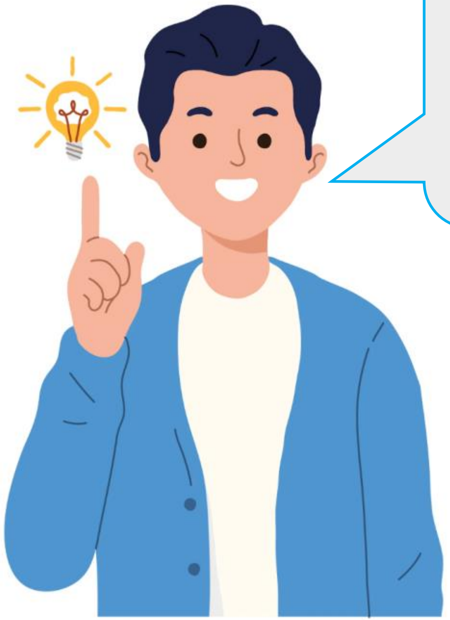
- Manually created test cases do not exercise the *physical conditions* that trigger the buggy behavior.



Physical condition 1: Creating a sharp corner through waypoints

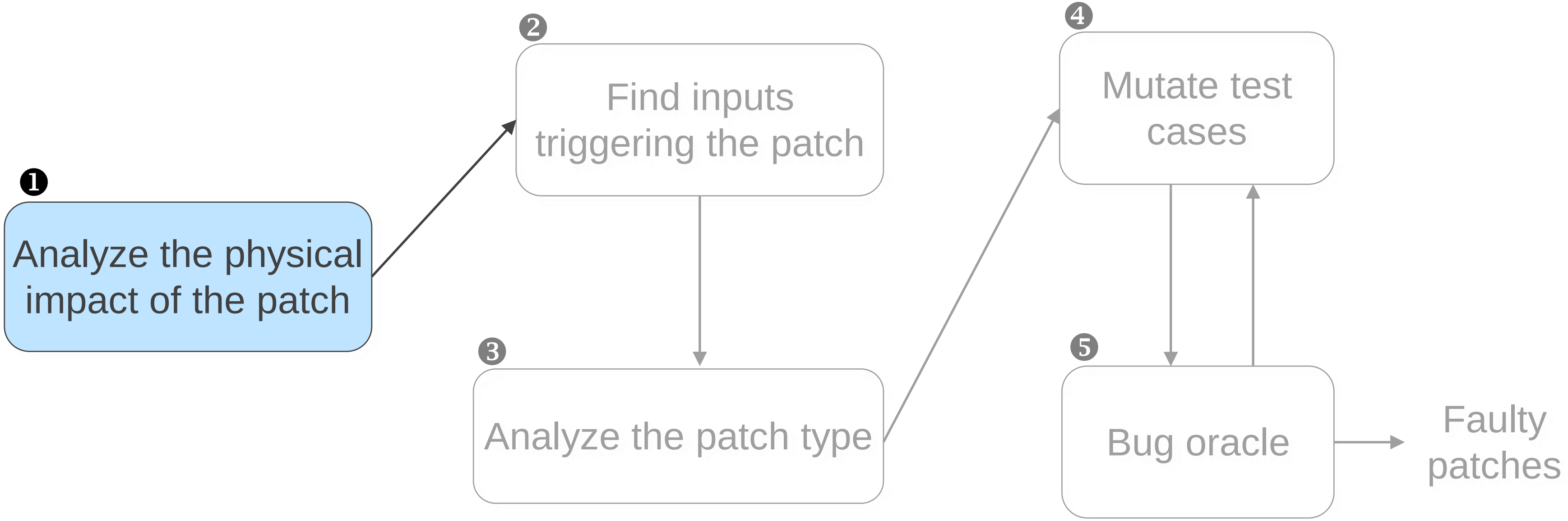
Physical condition 2: Setting a high ground speed (e.g., 10 m/s)

Main Idea of PatchVerif



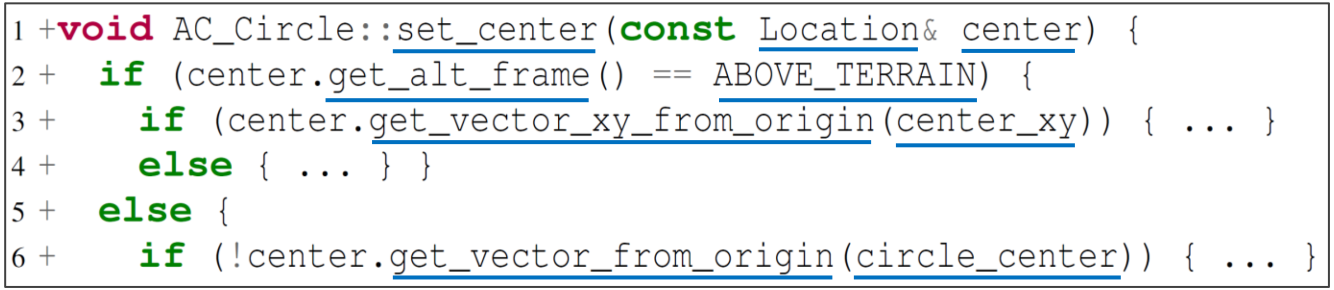
Let’s mutate test cases based on a given patch!

Overview of PatchVerif



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



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

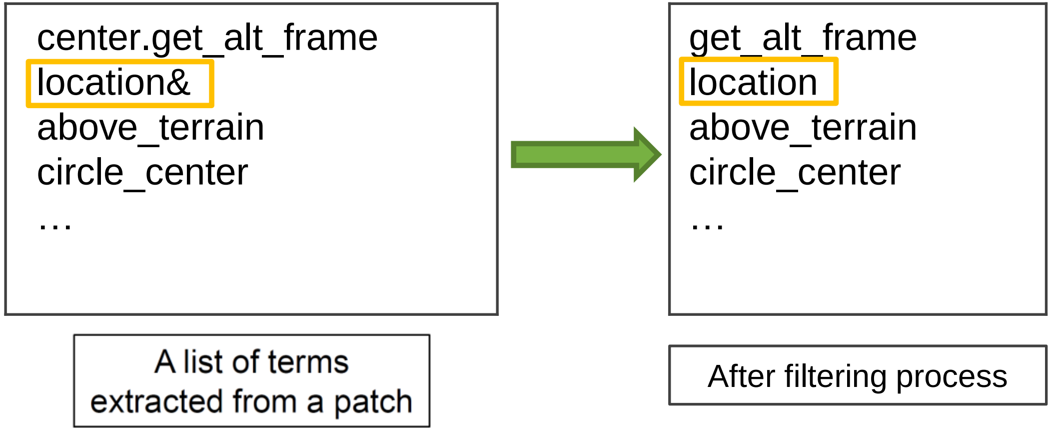
<A patch implementing terrain-following for the CIRCLE flight mode>

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

❶ Analyze Physical Impact of Patches

(2/4)

- 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
…

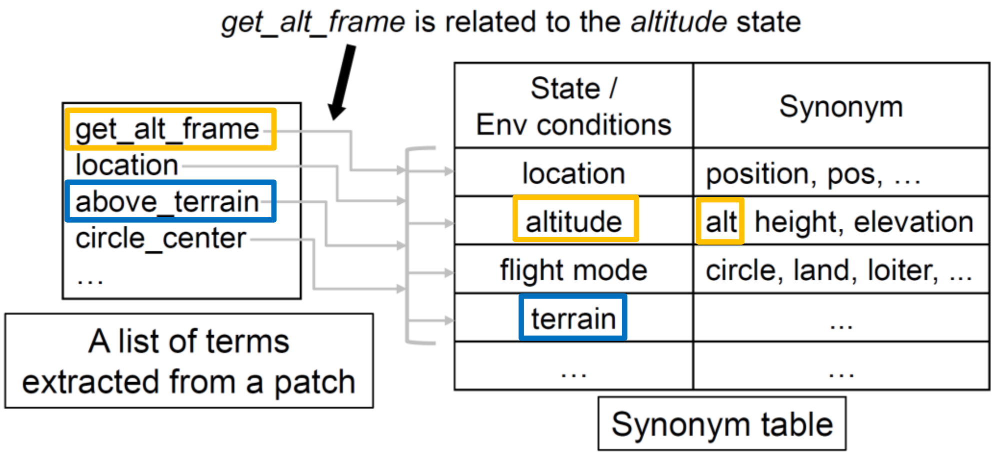
After filtering process

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 Physical_{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

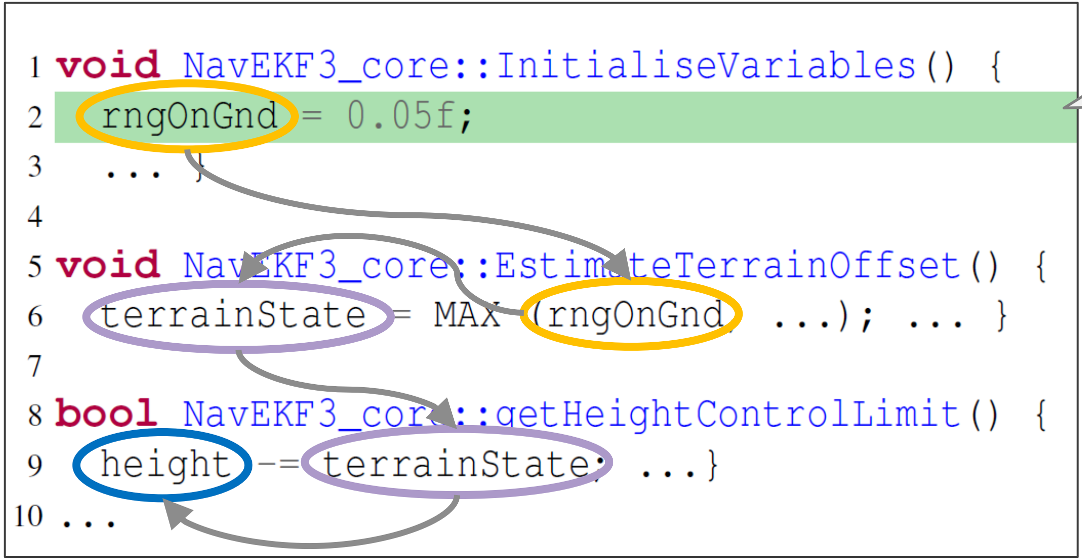
Step 3: Match the extracted terms with RV physical states and environmental conditions in the synonym table

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

A: Over-tainting issues

Example of Inter-dependency Problems

- Taint tracking is challenging
  - Due to high interdependency among variables in the RV software

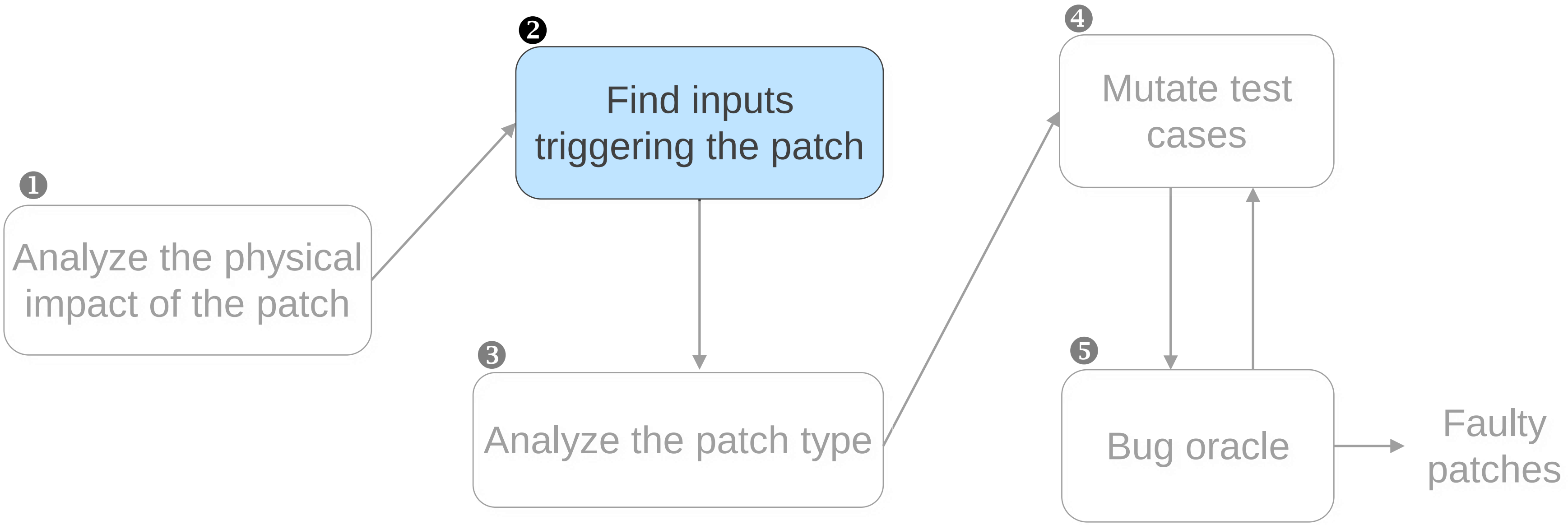


```
1 void NavEKF3_core::InitialiseVariables() {
2   rngOnGnd = 0.05f;
3   ... }
4
5 void NavEKF3_core::EstimateTerrainOffset() {
6   terrainState = MAX(rngOnGnd, ...); ... }
7
8 bool NavEKF3_core::getHeightControlLimit() {
9   height -= terrainState; ...}
10 ...
```

Developers add one line of patch code at line 2.

rngOnGnd’s **value is propagated to 43.6% of variables in the RV software**

Overview of PatchVerif



❶ Analyze the physical impact of the patch

❷ Find inputs triggering the patch

❸ Analyze the patch type

❹ Mutate test cases

❺ Bug oracle

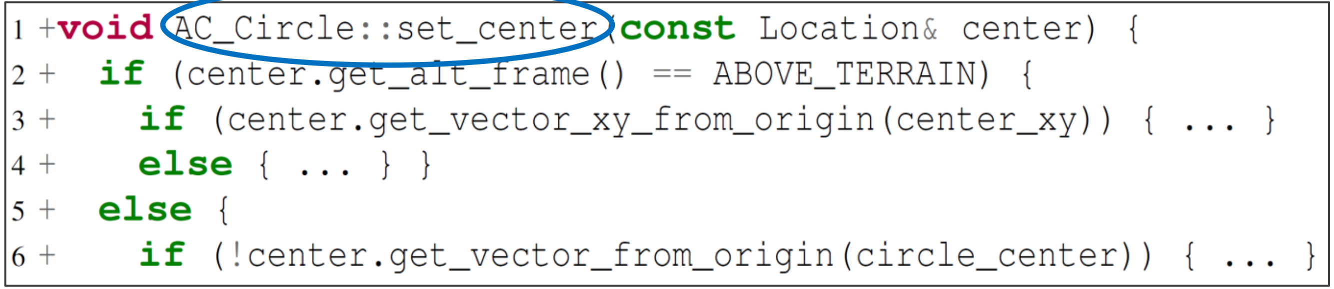
Faulty patches

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

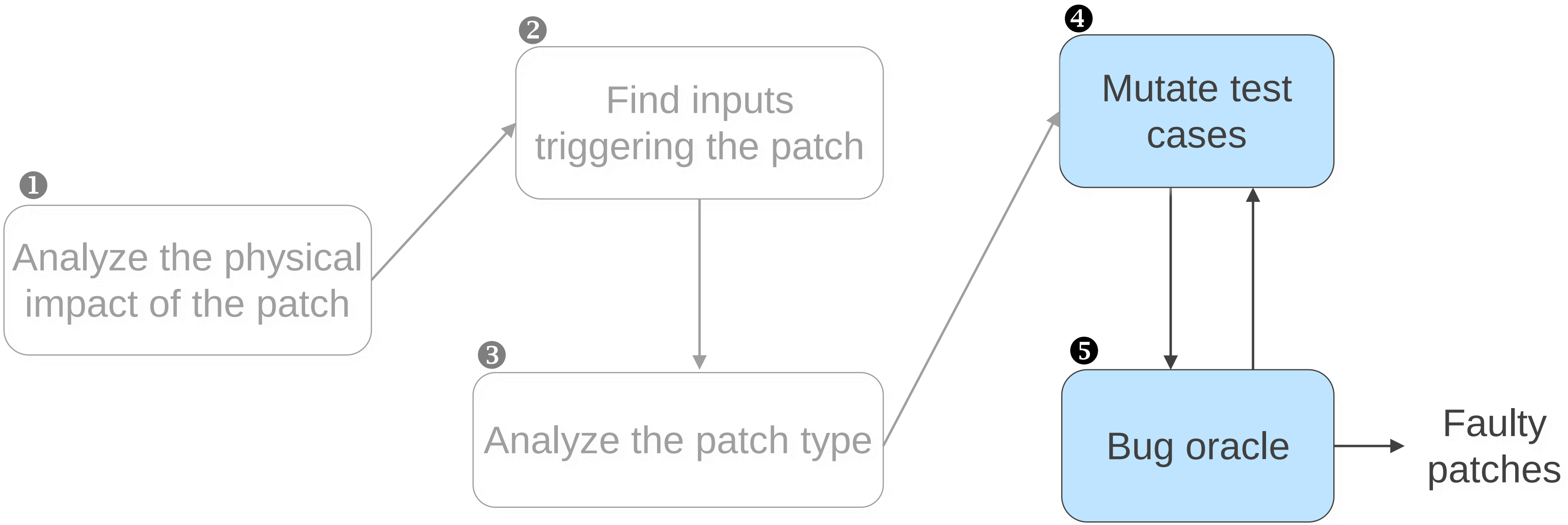
CIRCLE flight mode triggers the patch code snippet.



```
+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



❶ Analyze the physical impact of the patch

❷ Find inputs triggering the patch

❸ Analyze the patch type

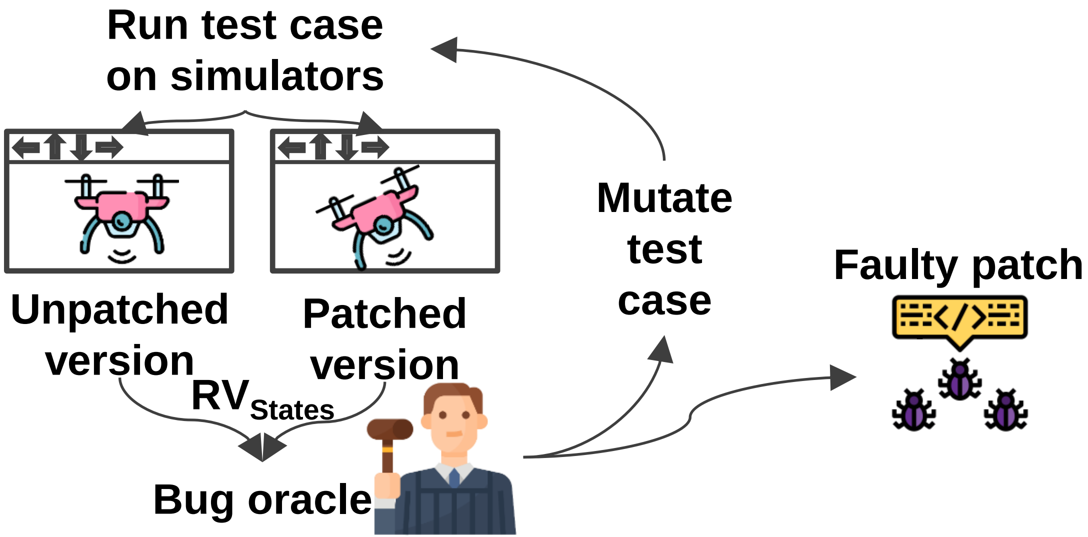
❹ Mutate test cases

❺ Bug oracle

Faulty patches

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

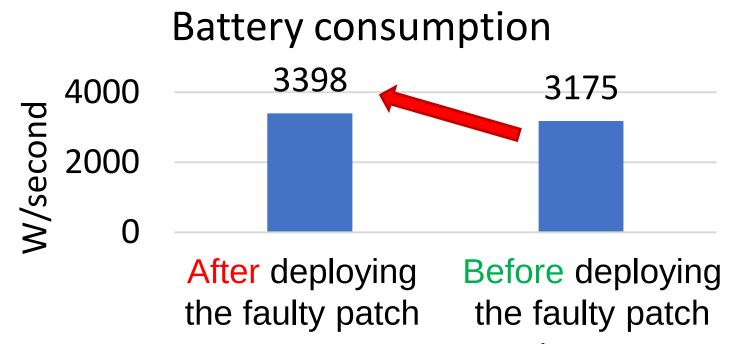


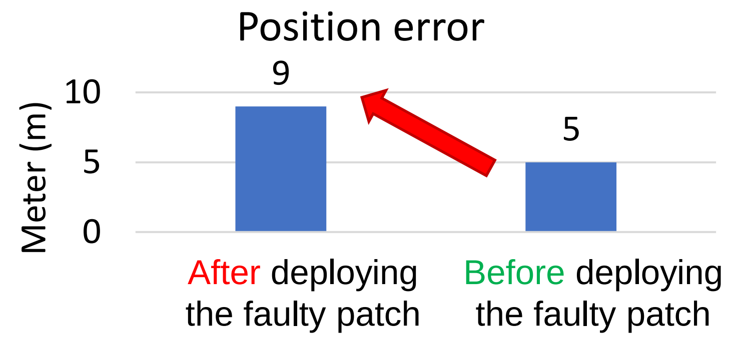
❹ Mutate Test Cases

- 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>

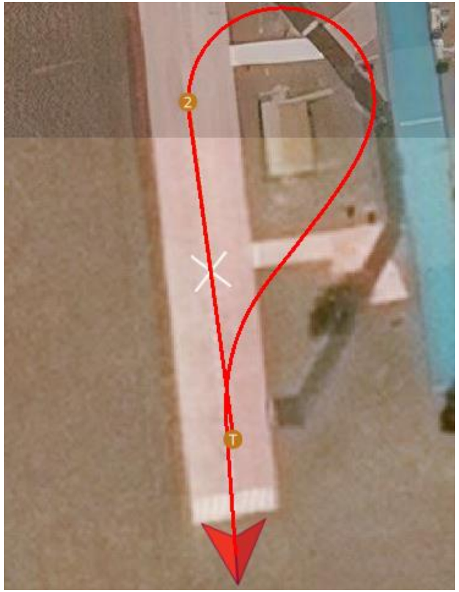




❺ Bug Oracle

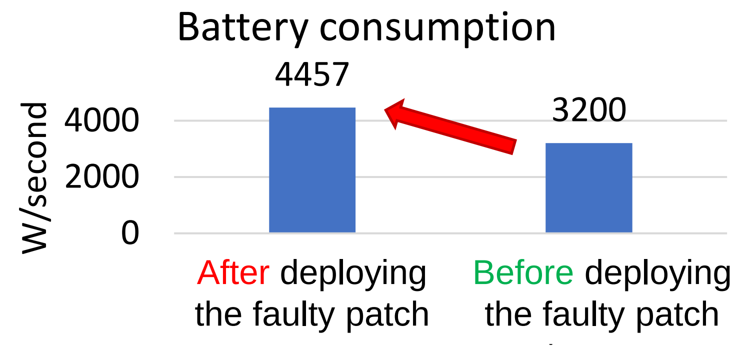
(3/3)

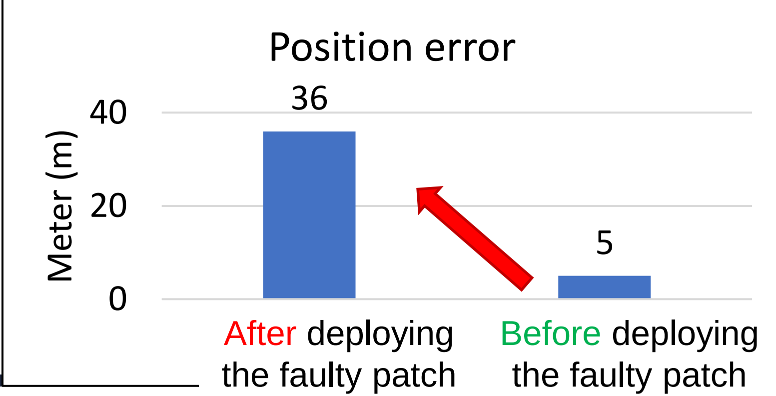
- Mutating the identified inputs to test the patch
  - Increasing the rover’s speed (**10 m/s**)



<**After** deploying the faulty patch>

<**Before** deploying the faulty patch>


Battery consumption


Position error

❺ Five Physical Invariants as Bug Oracles

- PatchVerif expects that a **correct patch should not**
  1) Increase mission completion time (Timeliness)
  2) Increase battery consumption (Efficiency)
  3) Increase position errors (Precise navigation)
  4) Increase instability (Stability)
  5) Cause a new error states (State consistency)

We manually extracted these invariants from a set of correct patches.

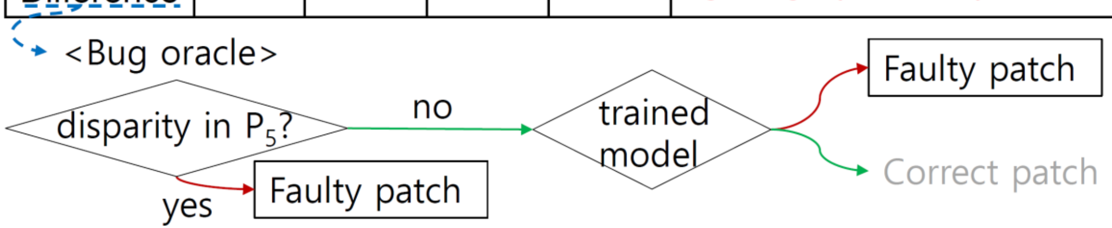
Challenge: A faulty patch can simultaneously cause *positive* and *negative* physical impacts.

Example: A patch *increases* stability errors and *decreases* position errors.

❺ Bug Oracle

- Our 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} *flight stage*={takeoff, flying, land}, …
$RV_{patched}$	600	71154	3.04	5.6	*gyro*={OK}, *gps*={OK} *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

Evaluation Results

- Dataset
  - 1,000 patches
    - 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

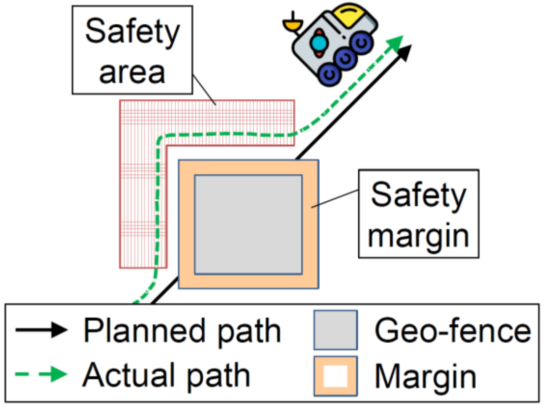
A Bug in Dijkstra Object Avoidance Algorithm

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

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

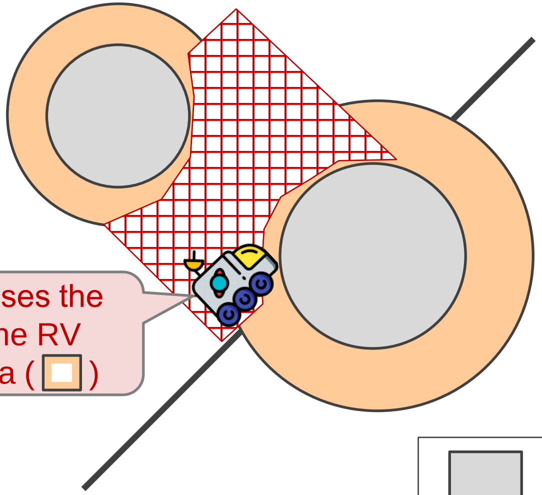
Case Study (Object Avoidance)

- 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

Takeaways from PatchVerif

- Writing patches for RV software is error prone.
  - Tracking patch-introduced behavioral modifications is hard.

- PatchVerif
  - Patch profiling
    - Extracting inputs related to a patch
  - Generate new test cases
    - By mutating patch-related inputs
  - Five physical invariants as bug oracles

Challenge 1: High-dimensional input spaces of RVs

Challenge 2: Defining correct behavior

Thank you for listening!

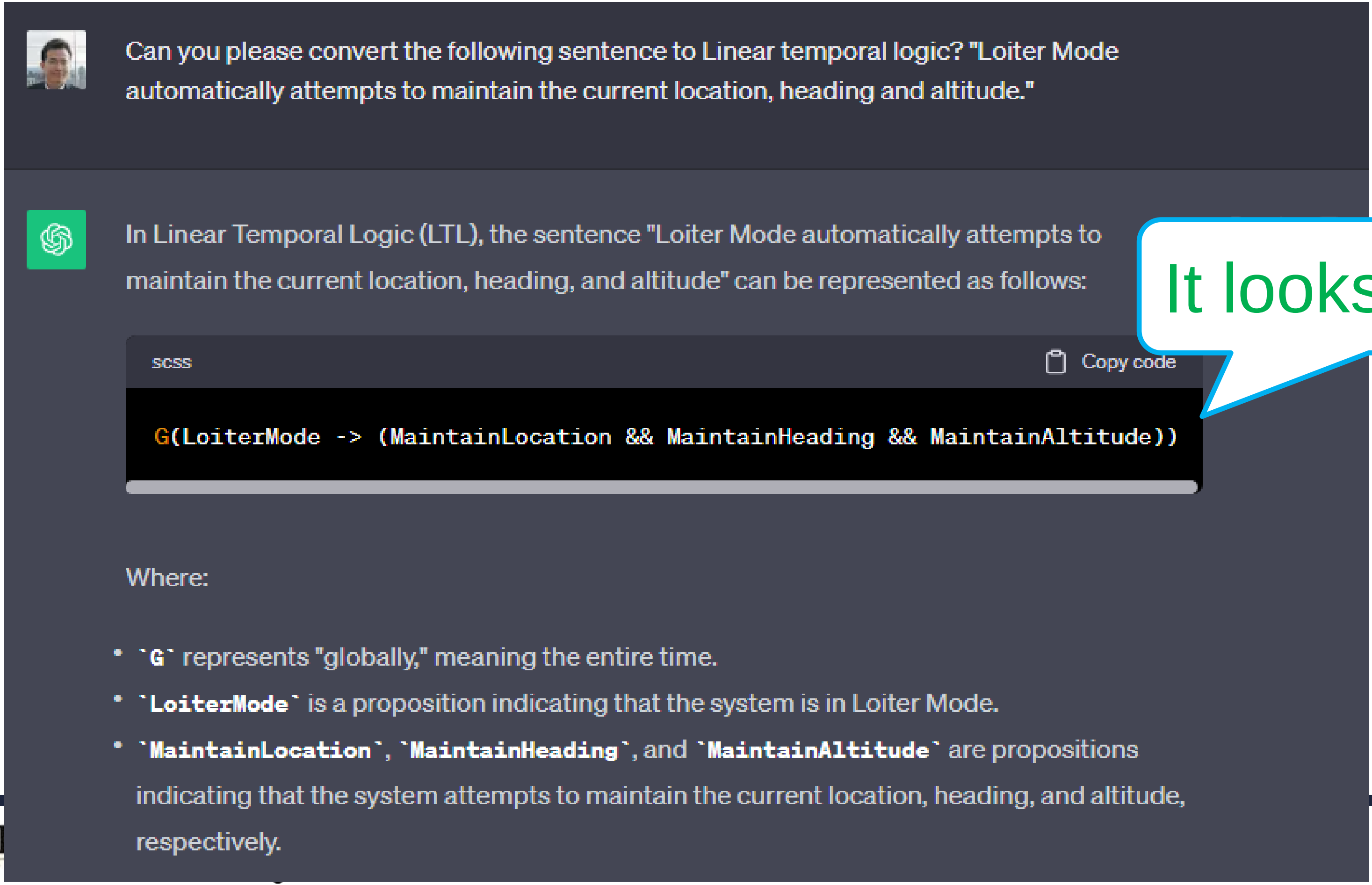
https://kimhyungsub.github.io

PURDUE UNIVERSITY

CERIAS

PurSecLab

Using LLMs to Extract Formulas (1)



Can you please convert the following sentence to Linear temporal logic? "Loiter Mode automatically attempts to maintain the current location, heading and altitude."

In Linear Temporal Logic (LTL), the sentence "Loiter Mode automatically attempts to maintain the current location, heading, and altitude" can be represented as follows:

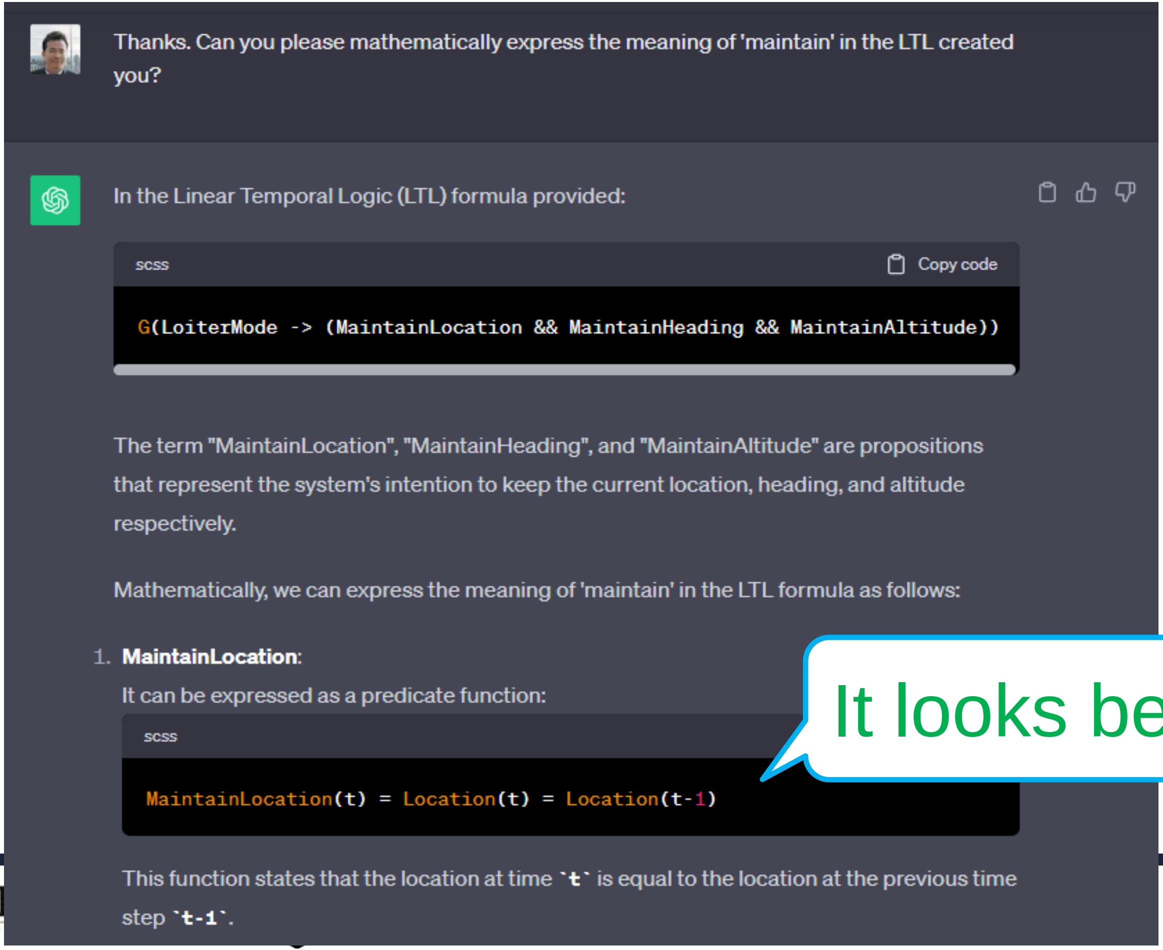
```scss
G(LoiterMode -> (MaintainLocation && MaintainHeading && MaintainAltitude))
```

Where:

- `G` represents "globally," meaning the entire time.
- `LoiterMode` is a proposition indicating that the system is in Loiter Mode.
- `MaintainLocation`, `MaintainHeading`, and `MaintainAltitude` are propositions indicating that the system attempts to maintain the current location, heading, and altitude, respectively.

It looks not bad!

Using LLMs to Extract Formulas (2)



Thanks. Can you please mathematically express the meaning of 'maintain' in the LTL created you?

In the Linear Temporal Logic (LTL) formula provided:

```
G(LoiterMode -> (MaintainLocation && MaintainHeading && MaintainAltitude))
```

The term "MaintainLocation", "MaintainHeading", and "MaintainAltitude" are propositions that represent the system's intention to keep the current location, heading, and altitude respectively.

Mathematically, we can express the meaning of 'maintain' in the LTL formula as follows:

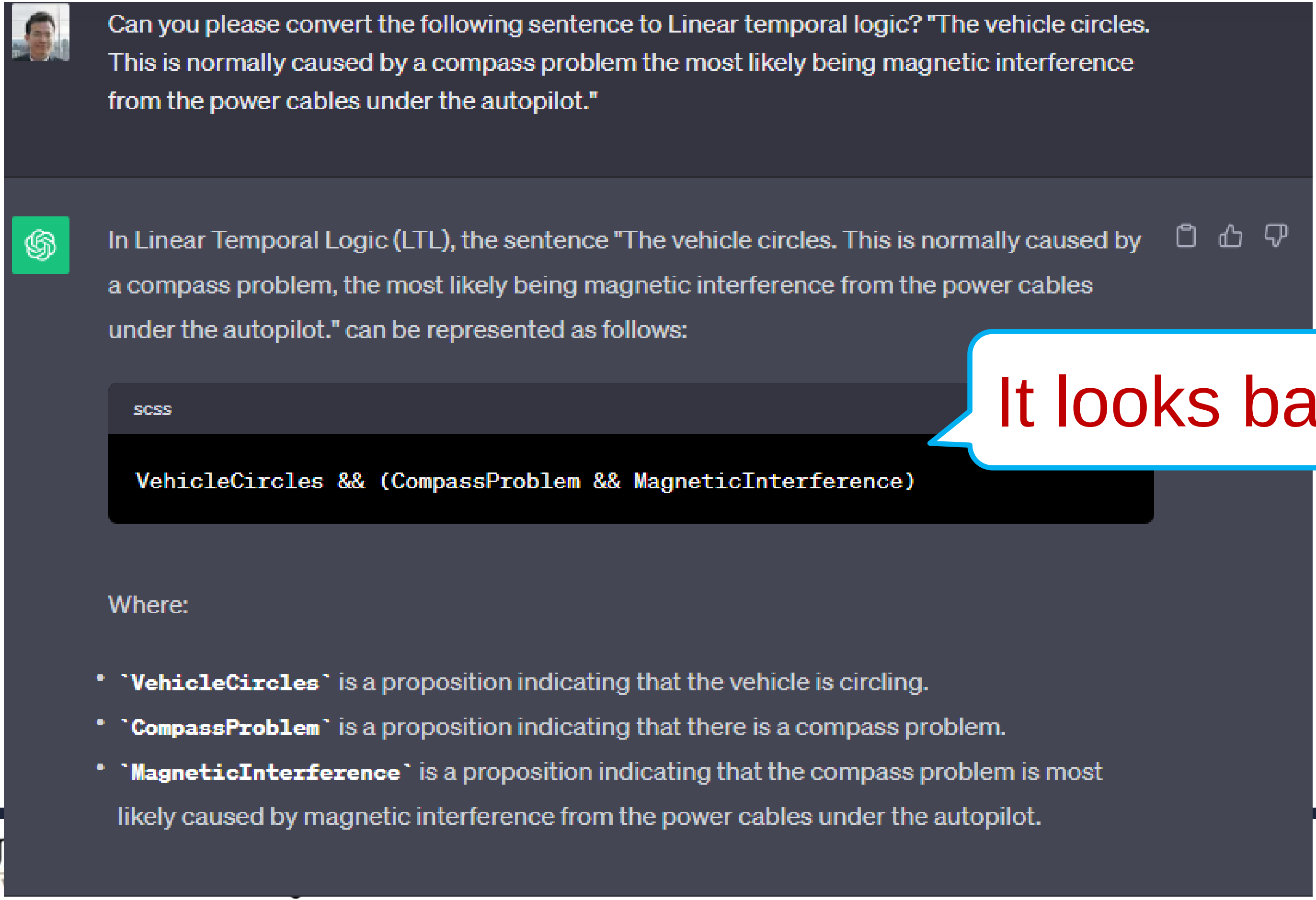
1. **MaintainLocation**:
   It can be expressed as a predicate function:

```
MaintainLocation(t) = Location(t) = Location(t-1)
```

This function states that the location at time `t` is equal to the location at the previous time step `t-1`.

It looks better!

Using LLMs to Extract Formulas (3)



Can you please convert the following sentence to Linear temporal logic? "The vehicle circles. This is normally caused by a compass problem the most likely being magnetic interference from the power cables under the autopilot."

In Linear Temporal Logic (LTL), the sentence "The vehicle circles. This is normally caused by a compass problem, the most likely being magnetic interference from the power cables under the autopilot." can be represented as follows:

```scss
VehicleCircles && (CompassProblem && MagneticInterference)
```

Where:

- `VehicleCircles` is a proposition indicating that the vehicle is circling.
- `CompassProblem` is a proposition indicating that there is a compass problem.
- `MagneticInterference` is a proposition indicating that the compass problem is most likely caused by magnetic interference from the power cables under the autopilot.

It looks bad!

Using LLMs to Extract Formulas (3)

Parsing natural language is still challenging.

LLMs do not know cause-and-effect relationship.

LLMs have no idea about sequences of behaviors.