DPFuzzer: Discovering Safety Critical Vulnerabilities for Drone Path Planners

Yue Wang[1,2,3], Chao Yang[1,2,3,*], Xiaodong Zhang[2,4], Yuwanqi Deng[1,3], JianFeng Ma[1,2,3]
[1]*State Key Laboratory of Integrated Services Networks (ISN), Xidian University, China*
[2]*Shaanxi Key Laboratory of Network and System Security, Xidian University, Xi'an, China*
[3]*School of Cyber Engineering, Xidian University, China*
[4]*School of Computer Science and Technology, Xidian University, China*
Email: ywang_86@stu.xidian.edu.cn, chaoyang@xidian.edu.cn, zhangxiaodong@xidian.edu.cn, dywq2021@163.com, jfma@mail.xidian.edu.cn

***Abstract*—State-of-the-art drone path planners enable drones to autonomously travel through obstacles in GPS-denied, uncharted, cluttered environments. However, our investigation shows that path planners fail to maneuver drones correctly in specific scenarios, leading to incidents such as collisions. To minimize such risks, drone path planners should be tested thoroughly against diverse scenarios before deployment. Existing research for testing drones to uncover safety-critical vulnerabilities is only focused on flight control programs and is limited in the capability to generate diverse obstacle scenarios for testing drone path planners.**

In this work, we propose *DPFuzzer*, an automated framework for testing drone path planners. *DPFuzzer* is an evolutionary algorithm (EA) based testing framework. It aims to uncover vulnerabilities in drone path planners by generating diverse critical scenarios that can trigger vulnerabilities. To better guide the critical scenario generation, we introduce *Environmental Risk Factor (ERF)*, a metric we propose, to abstract potential safety threats of scenarios. We evaluate *DPFuzzer* on state-of-the-art drone path planners and the experimental result shows that *DPFuzzer* can effectively find diverse vulnerabilities. Additionally, we demonstrate that these vulnerabilities are exploitable in the real world.

***Index Terms*—Drone, Testing, Fuzzing, Scenario Generation**

I. Introduction

The path planner is a fundamental component in many autonomous drone applications, for example search [1], [2], tracking [3]–[5], inspection [6], surveillance [7] and decentralized swarming [8], [9]. As these missions can be performed in cluttered and previously unknown environments, drones must be capable of autonomously navigating within obstacles. The path planner acts as a critical role in realizing this autonomy, showcasing extensive prospects [10]. Its primary functionality is to direct a drone fly along an obstacle-free route between different positions. As the drone path planner directly determines a drone's motion, the safety of the drone is highly relevant to it. Unfortunately, there is a lack of automated tools for testing drone path planners to evaluate their safety.

Research [11] shows that obstacles have the greatest impact on the planning algorithm. Meanwhile, our preliminary investigation finds that path planners can fail to appropriately maneuver drones in specific scenarios indeed, even resulting in severe consequences (e.g., drone crashes, and human injuries). For example, Figure1 illustrates such a scenario where the path planner aims to guide a drone to the destination G. Despite there are obstacle-free trajectories, for example P_a in the figure, the path planner still fails to handle the drone to avoid obstacles, resulting in a collision (i.e., P_b). We define the cause of misbehavior, where the path planner fails to guide the drone to its destination safely, as a safety-critical vulnerability in a path planner. Therefore, it is imperative to eliminate such vulnerabilities in path planners to mitigate severe consequences before deployment.

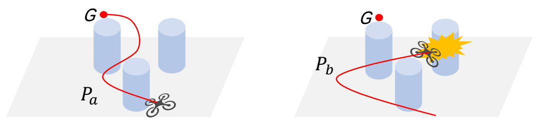


Fig. 1. The obstacle avoidance vulnerability.

Uncovering vulnerabilities is crucial in both the software engineering process [12], [13] and the safety assurance process [14], [15]. Identifying vulnerabilities facilitates developers to address or mitigate them preemptively before they result in severe consequences. Automated testing tools play a crucial role in this process. They can generate and execute numerous test cases, reporting those cases that trigger vulnerabilities, thereby significantly saving human effort. However, uncovering safety-critical vulnerabilities in cyber-physical systems (CPS) such as drones through automated testing is challenging. Unlike traditional computer programs, of which inputs are binary streams and vulnerabilities manifest as obvious execution failures (e.g., memory corruption). Tools such as AFL [16] can efficiently generate inputs and monitor execution to identify vulnerabilities in these traditional computer programs. Safety-critical vulnerabilities require specific physical inputs (e.g., temperature, obstacles) to trigger, and their anomalous behavior manifests in physical space (e.g., collision). Previous efforts [17]–[22] have given considerable efforts to automatically detect safety-critical vulnerabilities in drones through testing. For example, Kim [17] and Han [18] focus on detecting misconfigured parameters of flight control programs (e.g., PX4 [23] and Ardupilot [24]) which can result in the instability of the drone.

* Corresponding author.

Kim [19], [20] focus on generating factors such as wind, temperature and user commands which resulting flight control programs violating predefined safety policies (e.g., denying to release of the parachute under emergencies). Unfortunately, existing vulnerability detection techniques cannot detect such safety-critical vulnerabilities in drone path planners. As these works only focus on testing flight control programs, they are limited in generating factors, such as different wind speed and air pressure combinations, which are only related to vulnerabilities in flight control programs. The inputs of drone path planners are 3D surroundings, existing methodologies for testing flight control programs are not focused on its generation.

In this work, we propose *DPFuzzer*, an evolutionary algorithm (EA) based testing framework, to generate critical scenarios to uncover safety-critical vulnerabilities in drone path planners. EA-based scenario generation testing has shown their effectiveness in uncovering safety-critical vulnerabilities in automated driving systems (ADS) [25]–[30] of cars, such as Apollo [31] and Autoware [32]. However, these methods are tailored specifically to ADS. Due to the following inherent properties of these ADS testing frameworks, they cannot be applied to test drone path planners:

- Scenario generation: In ADS testing, scenarios are defined by domain-specific languages (DSLs) such as Openscenario [33] and AVUnit [34]. They are tailored to describe car scenarios (e.g., lanes and traffic lights) and can only be parsed for car simulators (e.g., Carla [35] and Lgsvl [36]). Frameworks for testing ADS are coupled with mutating these DSLs to mutate factors in scenarios. However, neither the scenarios described by these DSLs are suitable for testing drone path planners, nor do their corresponding simulators support drone simulation.
- Behavior monitor: ADS controls cars by sending steering, throttle, and brake commands to a car, which is a unique feature of ADS. However, drone path planners control drones also with unique commands (e.g., waypoint, velocity) which is different from ADS. Unluckily, frameworks designed for ADS monitor these exclusive control commands for scenario generation guidance or misbehavior detection.
- Mutation guidance: EA-based scenario generation utilizes metrics (or fitness function) to guide scenario mutation. However, these metrics measurements are designed to be coupled with cars' physical properties. For example, in the ADS testing framework, DriveFuzz [26], the metric calculation for scenario generation guidance involves the car's steering wheel angle, which does not exist in the drone. This makes the metrics unusable for scenario generation guidance in testing drone path planners.

The major challenge in our research is how to effectively generate scenarios which capable of triggering vulnerabilities while covering different vulnerability types. To address this challenge, we introduce a metric called ***environmental risk factor (ERF)***. ERF abstracts potential safety threats of a scenario, it assesses **global risk factor (GRF)** to abstract the likelihood of misbehavior happening in a scenario and assesses **obstacle risk factor (ORF)** to abstract the contribution of each obstacle in inducing a vulnerability trigger. The GRF is utilized for seed scenario selection guidance in each round of scenario generation. The ORF is utilized for mutation guidance to facilitate generating critical scenarios capable of covering various vulnerability types. We employed three path planners to evaluate *DPFuzzer*: `Ego-Planner` [37], `Ego-Planner-Swarm` [8] and `FUEL` [2]. We generated 969 critical scenarios that trigger vulnerabilities and categorized these vulnerabilities into 8 distinct types in `Ego-Planner`, 5 in `Ego-Planner-Swarm` and 7 in `FUEL`. Additionally, we demonstrated the exploitability of these vulnerabilities using a real-world commercial drone. Details are available in https://github.com/ywang-scholar/DPFuzzer.

This work makes the following contributions:

- We propose a novel testing framework *DPFuzzer*, which focuses on safety-critical vulnerabilities in drone path planners that have not been emphasized before.
- We propose the *environmental risk factor (ERF)* metric. It abstracts the potential threat of a scenario, which can efficiently guide the critical scenario generation.
- We evaluated *DPFuzzer* on different drone path planners and uncovered different types of vulnerabilities. Besides, we validated vulnerabilities in a real-world drone.
- The drone path planner is an essential component of many autonomous drone applications, this research also inspires the testing of these applications in future work.

II. Background

A. How Path Planners Cooperate with Drones

A basic drone primarily consists of a flight controller and peripheral hardware (e.g., propellers, GPS receivers). The flight controller is an embedded system consisting of hardware (e.g., Pixhawk series [38]) and software (e.g., PX4 [23], Ardupilot [24]). It is only responsible for some fundamental functionalities, such as maintaining the position and stability of the drone. Due to the limited computing resources and its specific role, the flight controller is not competent for complex tasks such as autonomous path planning. To enable path planning, the mainstream solution is to introduce a companion computer (e.g., Raspberrypi [39], NVIDIA Jetson [40]) to execute a path planner that commands the flight controller. For instance, a drone Fast-Drone-250 [41] equipped with PX4 [23] flight controller receives commands (e.g., position or velocity at each time step) [42] from its path planner (e.g., Ego-Planner-Swarm [43]). Path planners make decisions based on 3D information captured by depth sensors (e.g., LiDAR [44], [45], depth camera [46], [47]). Figure.2 illustrates how the flight controller cooperates with the path planner (Step ①-③). Given the destination, the path planner tries to generate a trajectory to the destination according to current surroundings (Step ①) and commands the flight controller to control the drone moves following the generated trajectory (Step ②). The flight controller tries to execute these commands loyally (Step

③). The above steps are repeated until the drone reaches the destination.

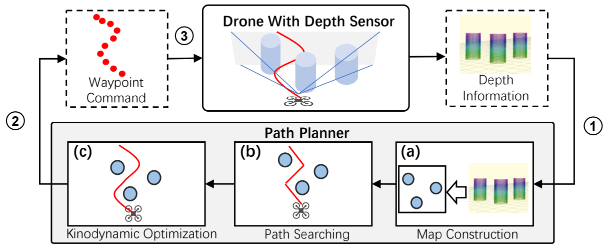


Fig. 2. General workflow of drone path planning.

B. General Workflow of Drone Path Planners

Generating obstacle-free trajectories to the destination is the core of the path planner. It is directly related to the safety of the drone as the drone moves following the generated trajectory. Steps $(a) - (c)$ in Figure 2 illustrate the general workflow of trajectory generation: (a) `Map construction` process involves converting depth information into a 3D map (e.g., TSDF [48], ESDF [49]) for subsequent path searching and kinodynamic optimization processes. (b) `Path searching` utilizes graph search algorithms (e.g., A*, RRT [50], JSP [51]) to find a general path to destination. This process takes multiple factors into consideration, such as the trade-off between shorter distance and safety. The general path only indicates general directions to the destination. It cannot be utilized to represent the trajectory since it does not fully consider the physical properties of the drone. (c) `Kinodynamic optimization` process generates a smooth trajectory based on the general path considering the physical properties of the drone.

The drone ultimately moves following the trajectory. Vulnerabilities in drone path planners are closely related to these trajectory generation processes. For example, (a) `map construction` inaccuracy can mislead the subsequent process, (b) `path searching` failures can result in unable to find the path to the destination or selecting an unsafe path that eventually leads to incidents, (c) `kinodynamic optimization` failures can generate a trajectory overlap with obstacles, ultimately leading to collision or stuck.

III. Design

A. Overview of DPFuzzer

Figure 3 illustrates the workflow of *DPFuzzer*. With the scenarios initialized (§III-B2), the **scenario generator** (§III-B) employs ***environmental risk factor (ERF)***, a metric we proposed, to generate scenarios. The **scenario executor** (§III-C) converts these generated scenarios into reality in a simulator and executes the path planner in the simulator. During the execution, the **misbehavior detector** (§III-D) monitors the status (e.g., position, velocity, and acceleration) of the drone at each time step and utilizes the test oracles to detect safety-critical misbehaviors. If misbehaviors are not detected, the **feedback engine** (§III-E) then assesses the ERF of each scenario for generation guidance. Through iteratively generating scenarios, *DPFuzzer* ultimately generates safety-critical scenarios that trigger vulnerabilities.

B. Scenario Generator

1) Scenario definition: A scenario is denoted by the tuple $\boldsymbol{s} = \langle \mathbb{B}, \mathbb{O}, \mathbb{R} \rangle$ as Figure 4a shows, where:

$\mathbb{B}$ is a finite set representing immutable boundary obstacles (e.g. walls) in the scenario (Figure 4b).

$\mathbb{O} = \{o_1, o_2, ..., o_n\}$ is a finite set describing mutable obstacles in the current scenario (Figure 4c). Each obstacle $o \in \mathbb{O}$ possesses attributes represented by a tuple denoted as $o = \langle pos, shape \rangle$, where pos represents the position of the obstacle and $shape$ represents its shape.

$\mathbb{R} = \langle \mathbb{R}_{grf}, \mathbb{R}_{orf} \rangle$ is a tuple representing *environmental risk factor* (ERF) metric of the scenario (Figure 4d). It is assessed by the **feedback engine** (§III-E). The $\mathbb{R}_{grf}$ evaluates the possibility of the whole scenario in inducing vulnerabilities triggering, while the $\mathbb{R}_{orf} = \{orf_1, orf_2, ..., orf_n\}$ evaluates the contribution of each obstacle in inducing vulnerabilities triggering. They are utilized for seed selection and mutation guidance in the scenario generation process respectively.

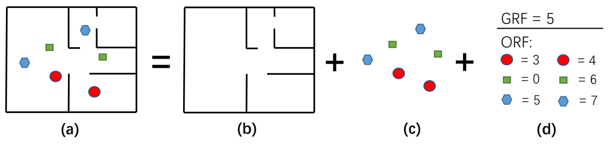


Fig. 4. Scenario definition. (a) scenario $\boldsymbol{s}$. (b) boundary obstacles definition $\mathbb{B}$. (c) mutable obstacles definition $\mathbb{O} = \{\boldsymbol{o_1}, \boldsymbol{o_2}, ..., \boldsymbol{o_n}\}$. (d) ERF definition $\mathbb{R} = \langle \mathbb{R}_{\boldsymbol{grf}}, \mathbb{R}_{\boldsymbol{orf}} \rangle$.

2) Scenario Initialization: Similar to conventional EA-based feedback-driven fuzzers [16], [52], *DPFuzzer* thrives when provided with initial inputs that are better suited. Given an immutable boundary obstacles definition $\mathbb{B}$, *DPFuzzer* generates the initial obstacle scenarios set $\mathbb{S} = \{s_1, s_2, ..., s_n\}$. *DPFuzzer* firstly executes the path planner in the scenario with only immutable boundary obstacles, i.e., $s_b = \langle \mathbb{B}, \varnothing, \mathbb{R} \rangle$, to observe the trajectory without obstacles (Figure 5a). Then, for each $\boldsymbol{s} \in \mathbb{S}$, *DPFuzzer* initialize $\mathbb{O}$, of which the obstacles are randomly arranged around the trajectory (Figure 5b). In that case, all of the generated obstacles are more likely to impact the path planner. As a comparison shown in Figure 5c, a completely random generated obstacles case, some of the obstacles are less likely to impact the path planner (i.e., the circular obstacle and the square obstacle). This process also initialize ERF attribute $\mathbb{R} = \langle 0, \{0, .., 0\} \rangle$.

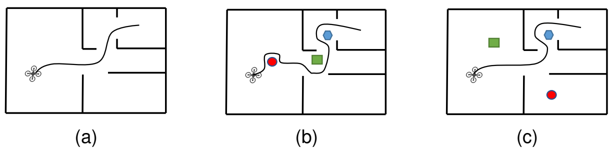


Fig. 5. Scenario initialization strategies. (a)Trajectory without obstacles. (b)Trajectory influenced by strategically generated obstacles. (c)Trajectory influenced by randomly generated obstacles.

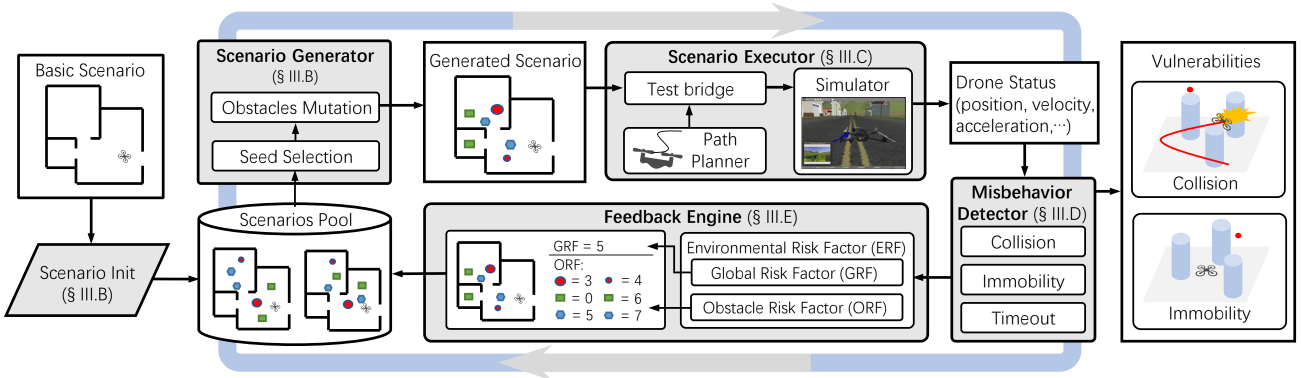


Fig. 3. Overview of *DPFuzzer*.

3) Scenario Generation: The process of scenario generation is shown as Algorithm.1. *DPFuzzer* mutates scenarios set $\mathbb{S} = \{s_1, s_2, ..., s_n\}$ to generate the new scenarios set $\mathbb{S}' = \{s'_1, s'_2, ..., s'_n\}$. It aims to iteratively mutate the scenarios set and finally generate scenarios that can induce vulnerability triggering. Scenario generation involves **seed selection** and **obstacle mutation**:

•***Seed Selection.*** Scenario $\boldsymbol{s} \in \mathbb{S}$ assigned with higher $\mathbb{R}_{\boldsymbol{grf}}$ is prioritized to be selected as seed (denoted as $\mathbb{P}$ in Line 2-3) since they are more likely to induce misbehavior (§III-E3).

•***Obstacle Mutation.*** The scenario generator generates new scenarios through mutating obstacle sets $\mathbb{O}$ of scenarios in seed set $\mathbb{P}$ (Line 4-20). To efficiently trigger vulnerabilities while covering different types of vulnerabilities, obstacle mutation involves two strategies as Figure 6 shows: (1)**Trigger First Strategy (TFS):** Slightly mutate position $\boldsymbol{o.pos}$ of obstacles assigned with higher ORF. As they typically directly induce accidents, slightly mutating them aims to capture the potential misbehavior near to happen. (2)**Explore First Strategy (EFS):** Significantly mutate position $\boldsymbol{o.pos}$ and shape $\boldsymbol{o.shape}$ of obstacles assigned with lower ORF. Although obstacles assigned with lower $\boldsymbol{orf}$ are typically not directly associated with triggering misbehaviors, they always act as a factor to influence the general path. Mutating them aims to increase the diversity of paths the drone travels to the destination, leading the drone to travel to the destination in different manners and ultimately increasing the variety of vulnerability types.

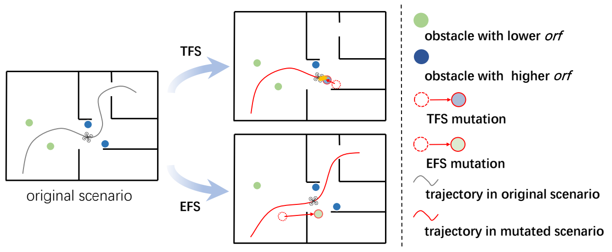


Fig. 6. Illustration of mutation strategies.

Algorithm 1: Scenario Generation

```
Inputs :
        Original scenarios set: $\mathbb{S}$.
        The number of population: $N$.
        The number of seeds: $M$.
        The number of obstacles to be mutated: $n$.
Output:
        Generated scenarios set: $\mathbb{S}'$.
1  Let $\mathbb{S}'$ be an empty set;
2  Sort set $\mathbb{S}$ in descending order according to $s.\mathbb{R}_{grf}$;
3  Select top M scenarios as seed: $\mathbb{P} \leftarrow \mathbb{S}[0:M]$;
   /* generate N scenarios                                  */
4  while $sizeof(\mathbb{S}') < N$ do
5      $p \leftarrow RandomSelect(\mathbb{P})$ ;
6      $strategy \leftarrow RandomSelect(\{TFS, EFS\})$ ;
       /* mutate m obstacles in p                            */
7      if $strategy$ is $TFS$ then
8          Sort $p.\mathbb{O}$ descending according to $\mathbb{R}_{orf}$;
9          foreach $o \in p.\mathbb{O}[0:m]$ do
10             $TFSMutation(o)$
11         end
12     end
13     else
14         Sort $p.\mathbb{O}$ ascending according to $\mathbb{R}_{orf}$;
15         foreach $o \in p.\mathbb{O}[0:m]$ do
16             $EFSMutation(o)$
17         end
18     end
19     Add $p$ into $\mathbb{S}'$;
20 end
21 return $\mathbb{S}'$ ;
```

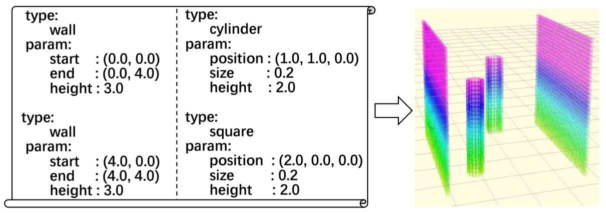
C. Scenario Executor

The scenario executor executes the path planner in given scenarios. Provided a scenarios set $\mathbb{S} = \{s_1, s_2, ..., s_n\}$, the scenario executor simulates the drone path planner in each scenario and records the drone status at each traversed position denoted as $\boldsymbol{DS} = \{ds_1, ds_2, ..., ds_n\}$. For each $s \in \mathbb{S}$, the scenario executor arranges obstacles in a simulator according to its boundary obstacles $\mathbb{B}$ and mutable obstacles $\mathbb{O}$ definition as Figure 7 shows. Then, it executes the path planner within the simulator and logs drone status $\boldsymbol{ds} = \langle \boldsymbol{P}, \boldsymbol{V}, \boldsymbol{A} \rangle$, where $\boldsymbol{P} = (p_1, p_2, ...p_n)$ is the sequence of the drone's position

Algorithm 2: Environmental Risk Factor Calculation

```
Inputs :
        Obstacle scenarios set: $\mathbb{S}$.
        The drone status: $DS$.
Output:
        Obstacle scenarios set $\mathbb{S}$ assigned with ERFs.
1 foreach $s \in \mathbb{S}$, $ds \in DS$ do
      /* calculate PR */
2     $PR \leftarrow PRCalculation(s, ds)$;   //§III-E1
      /* assign ORF for obstacles */
3     foreach $o \in s.\mathbb{O}$, $orf \in s.\mathbb{R}_{orf}$ do
4         $orf \leftarrow ORFCalculation(o, PR)$;   //§III-E2
5     end
      /* assign GRF for the scenario */
6     $s.\mathbb{R}_{orf} \leftarrow GRFCalculation(s)$;   //§III-E3
7 end
8 return $\mathbb{S}$;
```

at each timestep. $\boldsymbol{V} = (v_1, v_2, ...v_n)$ is the sequence of the drone's velocity at each timestep. $\boldsymbol{A} = (a_1, a_2, ...a_n)$ is the sequence of the drone's acceleration at each timestep.



```
type:                         type:
    wall                          cylinder
param:                        param:
    start  : (0.0, 0.0)           position : (1.0, 1.0, 0.0)
    end    : (0.0, 4.0)           size     : 0.2
    height : 3.0                  height   : 2.0

type:                         type:
    wall                          square
param:                        param:
    start  : (4.0, 0.0)           position : (2.0, 0.0, 0.0)
    end    : (4.0, 4.0)           size     : 0.2
    height : 3.0                  height   : 2.0
```

Fig. 7. Parsing scenario.

D. Misbehavior Detector

When the path planner fails to handle the drone appropriately in a given scenario, it can lead to incidents (e.g., collision). The misbehavior detector is designed to monitor $ds \in \boldsymbol{DS}$ at each timestep to check potential misbehaviors of the drone. We designed the following three test oracles oriented to the safety of the drone, which the misbehavior detector refers to for detecting misbehaviors:

- **Collision.** Collision is one of the most destructive incidents that can directly cause damage to drones. The collision monitor measures the drone's distance to obstacles. If the distance between the drone's center and the nearest obstacle is lower than a threshold (0.15 meters in this paper as we simulated a 0.3 meters wheelbase drone), it is considered misbehavior.
- **Immobility.** The drone stopped and hovered at a position that would lead to battery exhaustion and subsequent crash landing. The immobility monitor measures the duration when the drone is not moving. It is considered a misbehavior if this duration exceeds a threshold (10 seconds in this paper).
- **Timeout.** The inability of the drone to reach its destination within a constrained time may indicate misbehavior. Even if a collision or immobility does not happen, the drone might not be able to reach the destination for other reasons, such as being unable to find a valid path to the destination. The timeout monitor measures the duration taken by the drone to reach its destination. It is considered a misbehavior if this duration exceeds a threshold.

Upon detection of a misbehavior, the incident and the corresponding scenario are logged. False positives may arise due to a strict threshold setting. For instance, the drone may exceed the 25 seconds time limitation while still accomplishing its mission within 30 seconds, which should not be considered misbehavior. Thus, the detected misbehaviors should be replayed for manual inspection. Notably, a longer timeout configuration tolerance will extend the fuzzing time due to prolonged simulation, while a shorter tolerance may raise false positives, inducing more manual inspection workload. Our timeout configurations are empirically determined as a trade-off between inspection workload and fuzzing efficiency.

E. Feedback Engine

When no safety-critical misbehavior is detected, *DPFuzzer* analyzes the flight logs $\boldsymbol{DS} = \{ds_1, ds_2, ..., ds_n\}$ of each scenraio $\mathbb{S} = \{s_1, s_2, ..., s_n\}$ to provide feedback for mutation guidance. We proposed a metric named **environmental risk factor (ERF)** that abstracts the potential threat of a scenario in two aspects: (1) the **obstacle risk factor (ORF)** that abstracts the potential threat posed by each obstacle within the scenario for mutation guidance and (2) the **global risk factor (GRF)** that abstracts the potential threat posed by the whole scenario for seed selection guidance.

Given the scenario definition $\boldsymbol{s}$ and the status $\boldsymbol{ds}$ at each traversed position, the feedback engine firstly quantifies the **potential risk (PR)** at each traversed position. It evaluates various behaviors of the drone during the mission. These behaviors do not lead to incidents in the current scenario but tend to. The **PR** is later employed to evaluate ORF and GRF combined with the scenario definition $\boldsymbol{s}$. Algorithm.2 shows the process of **ERF** calculation.

1) PR Calculation: The *potential risk (PR)* denotes the likelihood of misbehavior happening at a traversed position $p \in ds.P$. A traversed position assigned with a higher PR indicates a higher likelihood of misbehavior occurrence. The potential risk in a traversed position is evaluated from multiple aspects. Inspired by metrics used to evaluate the performance of path planner [11] and features of some misbehavior we observed, we propose four types of risks to assess PR in a traversed position.

Reaction risk. The reaction risk measures the risk of a collision with obstacles that lie directly along the extension of the velocity vector. It was inspired by the architecture of popular drone path planners when we investigated their source codes. The path planner involves the emergency handling [53]–[56] process, such as the EMERGENCY_STOP [53] in Ego-Planner, which handles the appearance of a previously undetected obstacle or revoke a previously suboptimal decision that leads to a collision. A shorter time available for handling an emergency is risky. Reaction risk is quantified as the speed

(2-norm of velocity) divided by the distance to the nearest obstacle along the velocity vector direction. The formula is:

$$R_{reac,i} = \frac{\|\vec{v_i}\|}{d_{vel,i}} \tag{1}$$

where $\|\vec{v_i}\|$ represents the speed at traversed position i, and $d_{vel,i}$ represents the distance to the nearest obstacle along the extended velocity vector at traversed position i.

Hard turning risk. The drone moves following the trajectory generated by the path planner. A sudden lateral acceleration could cause a deviation from the intended trajectory when the projection of velocity in the forward direction of the drone is faster. This arises due to a portion of the thrust allocated for maintaining forward movement. When lateral acceleration is required (e.g., to avoid an obstacle), thrust must be reallocated for lateral. The faster the drone, the less thrust can be reallocated for lateral maneuverability. Consequently, the drone cannot promptly respond to the lateral maneuver. This issue also arises because the path planner fails to adequately consider the kinodynamic model of the drone and incorrectly allocates the velocity at positions to be traversed, even though the generated trajectory does not overlap with obstacles. This can be due to the kinodynamic properties of the drone is not well considered. The hard turning risk is quantified by considering the acceleration's projection in the direction perpendicular to the velocity and the current velocity. The formula is:

$$R_{turn,i} = \|\vec{v_i}\| \cdot \left\| \vec{a_i} - \vec{v_i} \cdot \frac{\vec{a_i} \cdot \vec{v_i}}{\|\vec{v_i}\|^2} \right\| \tag{2}$$

where $\vec{a_i}$ represents the acceleration at traversed position i, $\vec{v_i}$ represents the velocity at traversed position i.

Hard braking risk. The hard braking risk is designed based on some misbehaviors we observed. When the path planner tries a sudden braking, immobility often follows, such as vulnerability #3 and #5 shown in (§IV-B). The hard braking risk can be quantified by considering the acceleration's projection in the reversed direction parallel to the velocity. The formula is:

$$R_{brake,i} = \begin{cases} \|\vec{v_i}\| \cdot \left| \dfrac{\vec{a_i} \cdot \vec{v_i}}{\vec{v_i} \cdot \vec{v_i}} \right| & , \vec{a_i} \cdot \vec{v_i} < 0 \\ 0 & , \vec{a_i} \cdot \vec{v_i} \geq 0 \end{cases} \tag{3}$$

where $\vec{a_i}$ represents the acceleration at traversed position i, $\vec{v_i}$ represents the velocity at traversed position i.

Distance risk. The distance risk is straightforward, it measures the minimal distance to the nearest obstacle. A traversed position closer to obstacles is considered a higher distance risk. The formula is:

$$R_{dist,i} = \frac{1}{d_{min,i}} \tag{4}$$

where $d_{min,i}$ represents the distance to the nearest obstacle at traversed position i.

All of these above risk factors should be normalized referring to the maximum value in a round of generation to eliminate the bias, take the reaction risk for example:

$$NR_{reac,i} = \frac{R_{reac,i}}{Max\left(\left\{R_{reac,I}\middle|\ I = 1,2,3...n\right\}\right)} \tag{5}$$

PR calculation employs `Euclidean Norm` for risk factors fusion, it amplifies superior risks type:

$$PR_i = \sqrt{NR^2_{reac,i} + NR^2_{brake,i} + NR^2_{turn,i} + NR^2_{dist,i}} \tag{6}$$

2) ORF Calculation: The ORF evaluates the potential threat posed by an individual obstacle. The obstacle with a higher likelihood of directly influencing misbehavior is assigned a higher ORF. The feedback engine assesses the ORF of each obstacle based on the PR of each traversed position. The ORF of an obstacle is determined by the maximum potential impact it could make. The formula for calculating $orf_j \in \mathbb{R}_{orf}$ of obstacle $o_j \in \mathbb{O}$ is:

$$orf_j = Max\left(\left\{\frac{PR_i}{d_{j,i}}\ \middle|\ i = 1,2,3...n\right\}\right) \tag{7}$$

Where PR_i stands for the PR at position i, the $d_{j,i}$ stands for the distance from obstacle $\boldsymbol{o_j}$ to traversed position i.

3) GRF Calculation: The GRF indicates the potential threat posed by the entire scenario. The scenario more likely to induce misbehavior is assigned a higher GRF. The GRF is calculated based on the ORF of obstacles. The feedback engine utilizes the most threatening subset of obstacles to represent the potential threat of the scenario. As depicted in Figure 8, obstacles in cluster $c1$ are more likely to trigger misbehavior, thus assigned with higher ORF on average. Conversely, obstacles in cluster $c2$ are less likely to induce misbehavior and assigned lower ORF on average. Therefore, the feedback engine first clusters the obstacles and selects the cluster with the highest average ORF to represent the GRF. Obstacles are clustered as set $\boldsymbol{C} = \{c_1, ..., c_k\}$ according to the `euclidean distance` between them. Then, for each obstacle cluster $\mathbf{c}_i \in \boldsymbol{C}$, the risk estimator calculates the average ORF of obstacles denoted as set $\{corf_1, ..., corf_k\}$, and selects the maximal $corf$ as the GRF. The formula is:

$$GRF = Max\left(\{corf_i \mid i = 1,2,3...k\}\right) \tag{8}$$

where k is the number of clustering centers.

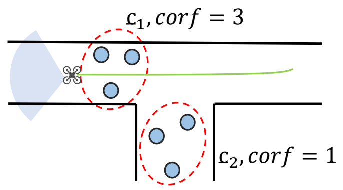


Fig. 8. Conceptual illustration of average ORF.

IV. Evaluation

A. Experiment Setup

- **Implementation.** We have implemented *DPFuzzer* with 2254 lines of Python and set it up on Ubuntu 20.04 with 8 GB memory and an AMD R7-5700 CPU.
- **Simulator.** We employ *Rviz* [57] with *so3-quadrotor-simulator* [58] plugin for simulation as Figure 10a shows. The simulated drone is assigned with a depth camera. The depth camera is capable of 120 degrees *horizontal field of view* (HFOV) and 4 meters of sensing range.
- **Targets.** We demonstrated *DPFuzzer* on two state-of-the-art open source drone path planners, the `Ego-Planner` [37] and the `Ego-Planner-Swarm` [59]. Besides, to demonstrate the scalability, we deployed *DPFuzzer* to evaluate `FUEL` [2] which significantly differ from the two path planners. `Ego-Planner` and `Ego-Planner-Swarm` are selected mainly for the following reasons: (1)Milestone: They made a milestone contribution to real-world drone path planning and many subsequently proposed drone path planners refer to their architecture design. (2)Openness: Both are popular in the open-source community and natively compatible with mainstream simulators. (3)Practicalness: They have demonstrated effectiveness in various models of drones and have been adopted by commercial drones [60], [61]. `Ego-Planner-Swarm` differs from `Ego-Planner` significantly in kinodynamic optimization algorithm. For `Ego-Planner`, it directly generates a unique trajectory based on the funded path [62]. However, this process in `Ego-Planner-Swarm` is more complex, it generates multiple distinctive trajectories, assesses them, and selects an optimal one [63]. The `FUEL` is a program based on path planning techniques, designed to autonomously construct a 3D map without prior knowledge of an unknown area. Compared to `Ego-Planner` and `Ego-Planner-Swarm`, FUEL should frequently adjust the drone's yaw direction to capture as much 3D information as possible when traveling to the destination, which makes it different.
- **Scenarios.** The evaluation employs simulated indoor scenarios (S1, S2) and alley scenarios (S3, S4) as basic scenarios for their typicality (e.g., rescue, delivery, and videography). Figure9 shows the plan view of scenarios utilized for testing. Generally, a more complex basic scenario will make it easier to find critical scenarios. However, the complexity of a basic scenario does not guarantee the diversity of vulnerability types. Testing within diverse basic scenarios enables developers to explore diverse vulnerability types.
- **Real-world Drone.** We demonstrated the exploitability of vulnerabilities in a commercial drone [60] in Figure10b, with a depth camera capable of 120 degrees HFOV and 4 meters of sensing range. It is equipped with Ego-Planner by default.

B. Research Questions

In the following, we conduct multiple experiments to answer the following Research Questions (RQs).

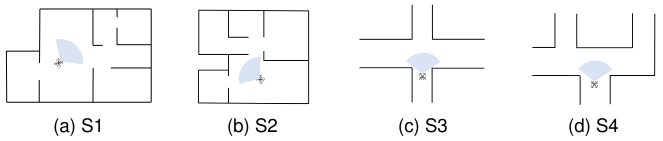


Fig. 9. Plan view of basic scenarios for testing.

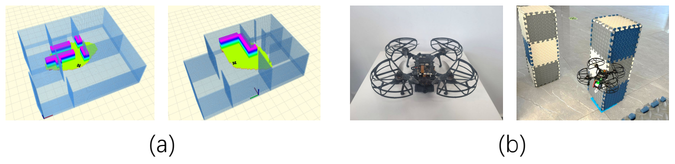


Fig. 10. (a)Simulator. (b)Real-world drone.

RQ1:What vulnerabilities can DPFuzzer reveals?

DPFuzzer finds 969 critical scenarios that lead to misbehavior, which can be classified into 8, 5, and 7 types of vulnerabilities in Ego-Planner, Ego-Planner-Swarm, and FUEL respectively. To answer *RQ1*, we summarize the in-depth analysis result of these vulnerabilities in Table I. In the table, the column **Vul.** represents the index of vulnerability type. The column **Impact** describes incidents the vulnerability could lead to. The column **Under Capability** describes the corresponding main causes of vulnerabilities. They are attributed as "under capabilities (UC)" according to the modules in path planners we introduced in §II-B. In the following, we will introduce vulnerabilities in path planners and their corresponding under capabilities. The videos and source code for replaying these vulnerabilities are available in [64].

Obstacle Avoidance UC. *OAUC* is highly relevant to the `kinodynamic optimization` modular introduced in §II-B. It arises due to the suboptimization of trajectory, which ultimately generates a trajectory that leads to a stuck or collision.

- Vulnerability #1 is that the drone collides with the obstacle on the side as shown in Figure 11. In this scenario, the drone is searching its path to destination (a). Before it reaches point P_a, constrained by its maximum perception range, the path planner plans to search along the trajectory T_a first. Upon the drone reaching P_a, updated data indicates that T_a is invalid. The path planner then commands the drone to promptly make a turning back maneuver to follow the alternate trajectory T_b (b). However, it fails to keep a safe distance from the lateral obstacle, resulting in a collision (c).

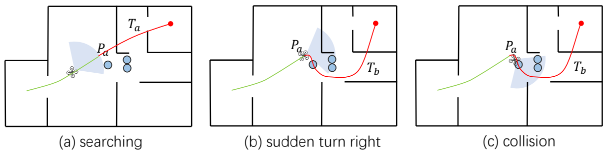


Fig. 11. Schematic diagram about vulnerability #1.

- Vulnerability #3 is that the drone becomes immobilized

TABLE I
SUMMARIZATION OF DISCOVERED VULNERABILITIES.

Vul.	Description	Impact*	Under Capability	Path Planner		
				EP*	ES*	FUEL
#1	Fails to avoid lateral collision	C	Obstacle Avoidance	✓	✓	✓
#2	Fails to avoid rear-end collision	C	Obstacle Avoidance	✓	✓	✓
#3	Finds a valid trajectory but become motionlessly near obstacles	C,I	Obstacle Avoidance	✓		✓
#4	Fails to find a valid trajectory to destination and circle in place	U	Path Searching	✓	✓	
#5	Fails to find a valid trajectory to destination and hover motionlessly	I	Path Searching	✓		✓
#6	Prioritize shorter trajectory over safety, leading to collision or immobility	C,I	Decision Making	✓	✓	✓
#7	Prioritize smooth trajectory over safety, leading to collision or immobility	C,I	Decision Making	✓		✓
#8	Imprecise map construction, leading to immobility or circle in place	I,U	Perception	✓	✓	✓

* **[Impact] C**: Collision. **I**: Drone becomes Immobile. **U**: Drone becomes Unstable.
* **[Path Planner] EP:** Ego-planner. **ES:** Ego-Planner-Swarm. ✓**:** Vulnerability found in the path planner.

although it successfully finds a feasible path to the destination as shown in Figure 12. In this scenario, the drone successfully finds a feasible path to its destination (b). However, when the drone reaches the midst of obstacles, the drone gets trapped. Although the path planner successfully finds a feasible trajectory, it refuses to continue traveling along the trajectory and the drone becomes immobilized.

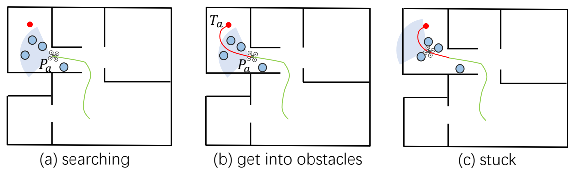


Fig. 12. Schematic diagram about vulnerability #3.

Path Searching UC. *PSUC* is highly relevant to the `path searching` modular introduced in §II-B. It refers to the path planner being unable to find a valid path to the destination although it exists. It can lead to the drone hovering motionlessly or circling in place, eventually resulting in battery depletion and subsequent crash landing.

- Vulnerability #5 is that the drone fails to find a valid trajectory to the destination and hovers motionlessly as shown in Figure 13. In the scenario, despite the existence of valid paths (e.g., path T_b in a), the path planner fails to find one and commands the drone to stay motionlessly (b). This vulnerability fundamentally differs from vulnerability #3, although they appear similar. For vulnerability #3, the `path searching` module successfully finds a valid path, but the `kinodynamic optimization` module is unable to generate a valid trajectory according to the path. While the vulnerability #5 is due to `path searching` module being unable to find a valid path.

Decision Making UC. *DMUC* is also closely related to the `path searching` module. Compared to *PSUC*, the key difference lies in the fact that, while *DMUC* can successfully find a path, it results in a suboptimal and unsafe path, which ultimately leads to misbehavior. This issue arises because the path searching algorithm fails to effectively evaluate the trade-offs among all the candidate paths.

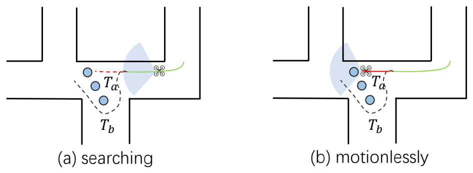


Fig. 13. Schematic diagram about vulnerability #5.

- Vulnerability #7 is that the drone prioritizes smooth trajectory over safety as shown in Figure 14. In the scenario, when the drone reaches position P_a (a), there are strategies: (1) the smooth but risky path (e.g., T_a in the diagram) or (2) the non-smooth but more safety path (e.g., T_b in the diagram). However, the path planner selected the smooth but risky one (b), colliding with an obstacle (c).

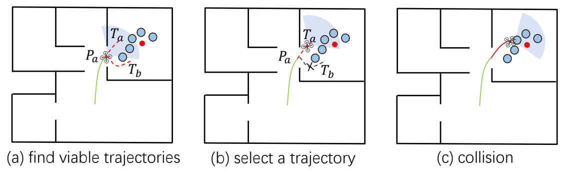


Fig. 14. Schematic diagram about vulnerability #7.

Perception UC. *PCUC* is highly relevant to the `map construction` modular introduced in §II-B. It arises when the path planner improperly processes sensor data and constructs a subaccurate map, misleading the subsequent path searching and kinodynamic optimization processes.

- Vulnerability #8 arises when the path planner fails to construct an accurate map. During map construction, the path planner samples the physical environment as discretized data (e.g., occupancy grid map [65]), with subsequent path searching and kinodynamic optimization processes relying on this map. However, due to limited sampling resolution, the map may fail to accurately represent the physical space. This imprecision can mislead the path planner into mistaking a sufficient gap for a narrower one, ultimately leading to immobility.

RQ2: How efficiency is DPFuzzer?

We evaluate the effectiveness of *DPFuzzer* in generating critical scenarios to cover the vulnerabilities and their corresponding types. Since there are no existing works focused on vulnerabilities in drone path planners, we conduct the comparison with Random method **(Rand)** and Distance-based GA method **(D-GA)**. **D-GA** is a modified version of *DPFuzzer* for ablation experiments, which takes the distance with obstacles for seed selection guidance. For all of these algorithms, we set the initial populations to 16 and the number of generations to 16, resulting in 240 test cases in one run. We initialize these algorithms with the same scenarios pool and run the comparison five times to reduce randomness. We evaluate `Ego-Planner`, `Ego-Planner-Swarm` and `FUEL` under all the scenarios shown in Figure 9. The results are summarized in Table II and its detailed data is available in [66]. Rows **D/R** shows the result about **D-GA** and **Rand**. Columns Δ shows the comparison result between **DPFuzzer** and **D-GA/Rand**.

DPFuzzer outperforms **Rand** in the number of generated critical scenarios ($\mathbf{116.26\%}$ advancement on average) and the coverage of vulnerability types ($\mathbf{103.33\%}$ advancement on average) on two planners. Particularly, compared to **Rand**, **DPFuzzer** demonstrates superior performance in S3 and S4 ($\mathbf{175.93\%}$ advancement on average). This is due to the less cluttered nature of S3 and S4, which makes it challenging for **Rand** to uncover vulnerabilities. Compared to **D-GA**, **DPFuzzer** demonstrates a $\mathbf{37.19\%}$ advancement on average in the number of generated critical scenarios and $\mathbf{57.42\%}$ advancement on average in the coverage of vulnerability types. The advancement is more significant in the coverage of vulnerability types. This is because **D-GA** solely focuses on a single risk (distance) to select scenarios for mutation, leading to its weak generalization capability.

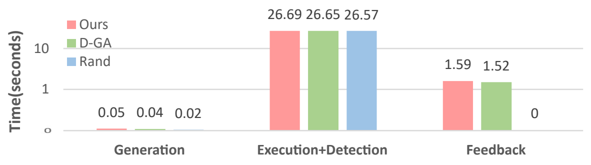


Fig. 15. Breakdown of the time consumption.

Figure 15 shows the average time consumption of each module in generating a scenario for testing Ego-Planner or Ego-Planner-Swarm. As execution (§III-C) and detection (§III-D) works in parallel, they are marked as **Execution+Detection** in the Figure 15. Time consumption for generation, execution, and detection are almost the same. Compared to **Rand**, **DPFuzzer** takes extra 1.59 seconds of time consumption. The difference in time consumption between **DPFuzzer** and **Rand** stems from the feedback module (§III-E). As **Rand** does not calculate metrics such as the ERF for feedback, its time overhead in feedback is 0 seconds. Nevertheless, the significant improvement in critical scenario generation and vulnerability types coverage overwhelmingly offset the extra 6.54% time consumption.

RQ3:Can these vulnerabilities trigger in the real world?

To show these vulnerabilities trigger in the real world, we model a real-world scenario into the simulator and then utilize *DPFuzzer* to generate safety-critical scenarios with obstacles. Subsequently, we arrange obstacles in the real world according to these generated scenarios (Figure 16). Given that only square-shaped obstacles are available in the real world, we only mutate the positions of obstacles during the scenario generation. *DPFuzzer* generates various critical scenarios that covers vulnerability #1, #3, #5 and #8. We discuss how vulnerability #1 triggers in this paper. Videos and other details about real-world experiments are available in [67].

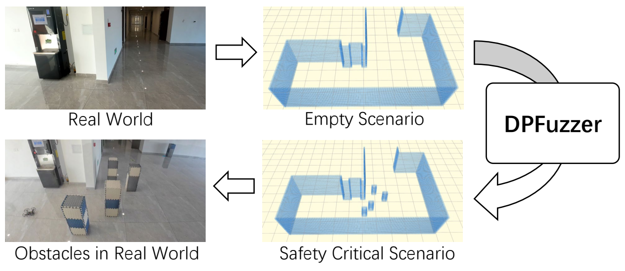


Fig. 16. Generating safety critical scenario in the real world.

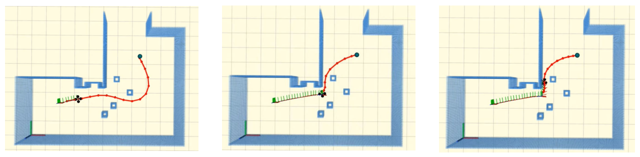
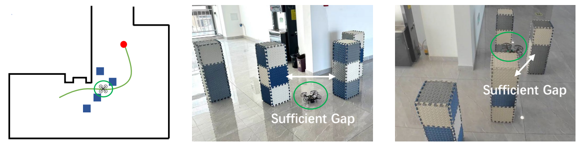
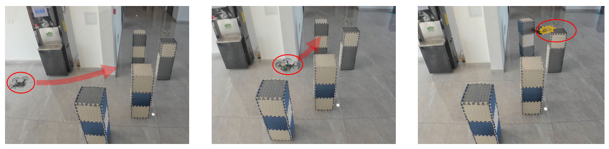


Fig. 17. Vulnerability triggering in simulator.



(a) An obstacle-free trajectory.



(b) Collision in real world.

Fig. 18. Vulnerability triggering in real world.

Figure 17 shows the vulnerability triggering in the simulator. The path planner initially finds a feasible path to the destination. However, when the drone reaches the corner, it suddenly turns left and grazes against the wall. Figure 18 shows the vulnerability triggering in the real world. Figure 18a shows that there are sufficient gaps for travel through. Figure 18b shows that, in the real world, the drone suddenly turns left at the corner and collides with an obstacle nearby. Due to the sensor noise and control output noise in the real world, the

TABLE II
COMPARISON IN THE NUMBER OF GENERATED CRITICAL SCENARIOS AND THE NUMBER OF COVERED VULNERABILITY TYPES.

Scenario	Path Planner	Alg.	The number of critical scenarios							The number of covered types						
			R1	R2	R3	R4	R5	Avg.	Δ	R1	R2	R3	R4	R5	Avg.	Δ
S1	Ego-Planner	Ours	11	12	8	12	9	10.4	+2.0(23.81%)/	5	5	4	6	6	5.2	+2.6(100.00%)/
		D/R	7/2	9/3	11/7	10/3	5/4	8.4/3.8	+6.6(173.68%)	3/2	3/3	3/3	2/2	2/2	2.6/2.4	+2.8(116.67%)
	Ego-Planner-Swarm	Ours	9	6	6	4	5	6.0	+1.8(42.86%)/	4	4	4	3	3	3.6	+1.2(50.00%)/
		D/R	3/2	2/5	4/4	4/1	8/4	4.2/3.2	+2.8(87.50%)	2/2	2/3	3/2	3/1	2/2	2.4/2.0	+1.6(80.00%)
	FUEL	Ours	13	10	11	14	11	11.8	+2.4(25.53%)/	6	5	5	6	5	5.4	+1.2(28.57%)/
		D/R	8/7	10/6	10/5	11/5	8/6	9.4/5.8	+6.0(103.45%)	4/4	4/3	5/4	4/3	4/4	4.2/3.6	+1.8(50.00%)
S2	Ego-Planner	Ours	10	8	12	14	7	10.2	+0.4(4.08%)/	6	5	6	5	6	5.6	+2.0(55.56%)/
		D/R	7/6	10/7	13/6	10/7	9/6	9.8/6.4	+3.8(59.38%)	4/3	4/3	3/3	3/2	4/3	3.6/2.8	+2.8(100.00%)
	Ego-Planner-Swarm	Ours	9	8	10	5	11	8.6	+0.2(2.38%)/	5	3	5	4	5	4.4	+1.4(46.67%)/
		D/R	8/7	6/5	7/10	9/2	12/8	8.4/6.4	+2.2(34.38%)	3/3	2/2	3/3	3/2	4/3	3.0/2.6	+1.8(69.23%)
	FUEL	Ours	9	13	10	10	13	11.0	+5.6(103.70%)/	5	4	6	5	6	5.2	+2.4(85.71%)/
		D/R	5/5	6/4	6/3	6/5	4/4	5.4/4.2	+6.8(161.90%)	2/4	3/3	3/2	3/3	3/3	2.8/3.0	+2.2(73.33%)
S3	Ego-Planner	Ours	2	2	3	2	4	2.6	+0.8(44.44%)/	2	2	2	2	3	2.2	+0.6(37.50%)/
		D/R	2/1	3/0	1/2	1/0	2/1	1.8/0.8	+1.8(225.00%)	2/1	2/0	1/2	1/0	2/1	1.6/0.8	+1.4(175.00%)
	Ego-Planner-Swarm	Ours	3	0	8	0	2	2.6	+0.6(30.00%)/	2	0	4	0	2	1.6	+0.2(14.29%)/
		D/R	1/0	3/0	3/0	1/0	2/3	2.0/0.6	+2.0(333.33%)	1/0	2/0	1/0	1/0	2/2	1.4/0.4	+1.2(300.00%)
	FUEL	Ours	9	5	6	6	9	7.0	+4.0(133.33%)/	6	5	5	5	5	5.2	+3.0(136.36%)/
		D/R	3/1	3/2	2/3	3/3	4/2	3.0/2.2	+4.8(218.18%)	3/1	2/2	1/2	2/3	3/2	2.2/2.0	+3.2(160.00%)
S4	Ego-Planner	Ours	9	8	4	8	7	7.2	+1.4(24.14%)/	5	3	4	5	2	3.8	+1.0(37.71%)/
		D/R	4/5	8/3	9/1	6/3	2/4	5.8/3.2	+4.0(125.00%)	3/2	3/1	4/1	3/2	1/2	2.8/1.6	+2.2(137.50%)
	Ego-Planner-Swarm	Ours	4	6	1	3	4	3.6	+0.6(20.00%)/	3	3	1	3	2	2.4	+0.4(20.00%)/
		D/R	2/0	3/2	2/3	3/2	5/0	3.0/1.4	+2.2(157.14%)	1/0	3/1	1/2	2/1	3/0	2/0.8	+1.6(200.00%)
	FUEL	Ours	7	5	8	9	5	6.8	+4.0(142.86%)/	5	3	4	5	4	4.2	+1.8(75.00%)/
		D/R	4/3	2/5	3/2	4/0	1/3	2.8/2.6	+4.2(161.54%)	4/2	2/3	2/2	3/0	1/3	2.4/2.0	+2.2(110.00%)

[*] **D/R:** D-GA/Rand, the result of the Distance-based GA method and the Random method, respectively.
[*] **Δ:** The result *DPFuzzer* compared to the Distance-based GA method and the Random method, respectively.

behavior observed in the simulation is not exactly the same as those in the real world [68]. Nonetheless, both of the causes were all sudden left turns at the corner and failure to avoid the obstacle. The experiment also shows that these vulnerabilities are exploitable. Attackers can meticulously place seemingly harmless obstacles according to scenarios generated by *DP-Fuzzer* to induce an accident deliberately.

Threats to Validity. Due to the inherent nature of simulation-based testing, there are threats to the validity. Some of the generated critical scenarios may induce incidents only in the simulation, while others may induce incidents solely in the real world.

Generally, in the real world, drone path planners are typically deployed on ARM-based embedded computers carried by the drone. These embedded computers are generally limited in computational capabilities compared to those x86-based in the simulation. Since path planning is a time-sensitive task, the limited computational resources can lead to incidents in scenarios that perform normally in simulation testing. Besides, some scenarios may lead to incidents solely in the simulator due to inconsistencies between the simulation and the real world [68]. Nonetheless, generating such scenarios remains valuable, as they indicate the patterns in which incidents are triggered. This still helps developers understand the mechanisms behind incidents and improve path planners to mitigate similar issues.

In our experiment, we employed *Rviz* with *so3-quadrotor-simulator* plugin for simulation. It simulates an ideal drone that always faithfully executes the commands from the path planner. However, in real deployment, the drone is not ideal; it is managed by a flight controller (e.g., PX4 [23]). The path planner commands the flight controller, and the flight controller attempts to control the drone to behave as the path planner intends. We found that the flight controller cannot always handle the drone to execute the commands from the path planner accurately and promptly. In some scenarios, even though the path planner functions correctly, it can still lead to incidents due to inaccuracies and hysteresis in the command execution. Since this research focuses solely on the path planner, incidents caused by coordination with the flight controller are beyond the scope of this study. Our selection of the simulator significantly mitigates such issues. However, such vulnerabilities are critical and threatening, we consider uncovering such vulnerabilities to be important future work.

V. DISCUSSION

•***Mitigation.*** Vulnerabilities are a critical issue in the software engineering process [12], [13] and safety assurance process [14], [15]. *DPFuzzer* can automatically generate and report test cases that trigger vulnerabilities, saving the efforts of human testing. Similar to other safety-critical vulnerabilities in drones (e.g., [17], [19]), the root causes of vulnerabilities identified by *DPFuzzer* primarily stem from source code and parameter configuration. We attributed the cause of vulnerabilities to different types of *under capabilities*, which can facilitate vulnerability fixing. For instance, **Path Searching UC** can be mitigated by adjusting path searching related parameters. Demonstrations of vulnerability mitigation are available in [69]. Methodologies for vulnerability mitigation are critical in systematic processes such as software engineering processes and safety assurance processes. These findings motivate us to explore automated methods for vulnerability mitigation in future work.

•***Scalability and Applicability.*** *DPFuzzer* generates physical scenarios to test the drone path planner. It monitors the drone status during the test to detect misbehavior and drives scenario generation for the next round of testing. Therefore, to deploy *DPFuzzer*, the prerequisite is that the target path planner can be simulated in a simulator, which can (1) parse scenario descriptions into reality and (2) provide drone status during the test. The latter (2) has been satisfied by many mainstream simulators, such as Rviz [57], Gazebo [70] and Airsim [71]. The former (1) is currently more challenging, as tools like DSLs for parsing scenarios in simulators, such as Openscenario [33] and AVUnit [34] in car simulation, are lacking in drone simulation. In this research, we have developed a tool similar to DSLs and have adapted it to the *Rviz* simulator. To summarize, currently, the drone path planner that supports simulation in *Rviz* can be easily adapted to *DPFuzzer*. In future work, developing tools DSLs and adapting them to different simulators for drones is an important issue, which facilitates the scalability of drone testing tools.

•***Safety Requirement.*** The ERF metric and oracles are designed with consideration of the safety requirement of drones. In this research, the safety requirement is considered according to the primary objective of the drone path planner: guiding the drone to its destination without collision. However, the safety requirement changes across different application scenarios. For instance, in tracking mission [3], [4], the safety requirement of following the moving target while keeping a safe distance should be considered. In swarming missions [8], [9], the safety requirement of keeping a distance between drones must be considered. Although, both of the two applications are based on path planning techniques, however, their safety requirements change due to their different application scenarios and mission objective. Unlike ADS, of which the application scenarios and the primary mission objective are relatively constant, the application scenarios and the primary mission objective for drone path planners are variable. In future work, we aim to extend the metrics and oracles for uncovering vulnerabilities of drone path planners in specific applications.

VI. Related Work

•**Safety-critical vulnerabilities in drones.** Existing researches on drone testing mainly focus on parameter configuration vulnerabilities [17], [18], [21], logical vulnerabilities [19], [20], [22], semantic correctness vulnerabilities [72]. For example, T. Kim et al. [17] highlight that incorrect parameter configurations in the control program can induce misbehavior. They introduce a half-interval search method to detect such vulnerabilities. H. Kim et al. [19] propose that flight control programs may violate predefined security policies under specific conditions and they introduce PGFuzz to uncover vulnerabilities. Although existing research proposes methodologies for identifying drone vulnerabilities from various perspectives, these studies primarily focus on testing flight control programs. Their focus limits their capacity to consider only flight control program related factors, which are not directly related to the drone path planner.

•**EA-based scenario generation for ADS testing.** EA-based scenario generation has been widely adopted in testing autonomous driving systems (ADS) of cars. Some works mainly focus on generating scenarios that lead to collision [25], [26], [30], [73], [74]. For example, AVFuzzer [25] utilizes the distance from ego-car to other vehicles to guide the scenario mutation for fuzzing. Similarly, DriveFuzz [26] mutates NPCs (e.g., pedestrians, other vehicles, puddles) that affect the frictional force and weather conditions to trigger safety-critical incident of car controlled by ADS. In addition, some works focus on policy violations [29], [75]. For example, LawBreaker [29] is an EA method to generate different scenarios leading to violations of traffic laws. Our methodology is inspired by the EA-based scenario generation as the drone path planner also operates in 3D scenarios similar to ADS. However, drones and cars are fundamentally distinct systems in numerous aspects (e.g., the kinodynamic model, scenarios, mission objectives). Consequently, the methods for ADS cannot be applied to drone path planners. The design of *DPFuzzer* considered the exclusive properties of drone path planners.

VII. Conclusion

In this paper, we emphasized the safety-critical vulnerabilities in drone path planners and proposed *DPFuzzer*, a novel framework for testing drone path planners, to uncover safety-critical vulnerabilities by generating various scenarios with obstacles. To better adapt to scenario generation, we proposed *Environmental Risk Factor (ERF)* for generation guidance. *DPFuzzer* can efficiently generate critical scenarios which capable of covering different vulnerability types. We implemented and evaluated *DPFuzzer* in 3 path planners, `Ego-Planner`, `Ego-Planner-Swarm` and `FUEL`. *DPFuzzer* was able to reveal 8 types of vulnerabilities in `Ego-Planner` with 352 safety critical scenarios, 5 types of vulnerabilities in `Ego-Planner-Swarm` with 250 safety critical scenarios and 7 types of vulnerabilities in `FUEL` with 367 safety critical scenarios. We validated the exploitability of vulnerabilities in a commercial drone.

VIII. Acknowledgment

We would like to thank all the anonymous reviewers for their insightful and constructive comments.

This work was supported by the Key Program of the National Natural Science Foundation of China (Grant No.62232013), the Foundation for Innovative Research Groups of the National Natural Science Foundation of China (Grant No.62121001), the Technology Innovation Leading Program of Shaanxi (Program No.2024QCY-KXJ-171), the China 111 Project (Grant No.B16037), the National Natural Science Foundation of China (Youth Program, Grant No.62402367), and the Fundamental Research Funds for the Central Universities (Grant No.20101247556).

References

[1] W. Tabib, K. Goel, J. Yao, C. Boirum, and N. Michael, "Autonomous cave surveying with an aerial robot," *IEEE Transactions on Robotics*, vol. 38, no. 2, pp. 1016–1032, 2021.

[2] B. Zhou, Y. Zhang, X. Chen, and S. Shen, "Fuel: Fast uav exploration using incremental frontier structure and hierarchical planning," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 779–786, 2021.

[3] DJI, "How to film like a pro: Dji drone "activetrack" video tutorial (2020)," Dec 2017. [Online]. Available: https://store.dji.com/guides/film-like-a-pro-with-activetrack/

[4] B. Jeon, Y. Lee, and H. J. Kim, "Integrated motion planner for real-time aerial videography with a drone in a dense environment," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 1243–1249.

[5] Z. Han, R. Zhang, N. Pan, C. Xu, and F. Gao, "Fast-tracker: A robust aerial system for tracking agile target in cluttered environments," in *2021 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2021, pp. 328–334.

[6] H.-A. Langåker, H. Kjerkreit, C. L. Syversen, R. J. Moore, Ø. H. Holhjem, I. Jensen, A. Morrison, A. A. Transeth, O. Kvien, G. Berg *et al.*, "An autonomous drone-based system for inspection of electrical substations," *International Journal of Advanced Robotic Systems*, vol. 18, no. 2, p. 17298814211002973, 2021.

[7] E. Unlu, E. Zenou, N. Riviere, and P.-E. Dupouy, "An autonomous drone surveillance and tracking architecture," in *2019 Autonomous Vehicles and Machines Conference, AVM 2019*, vol. 2019, 2019, pp. 35–1.

[8] X. Zhou, X. Wen, Z. Wang, Y. Gao, H. Li, Q. Wang, T. Yang, H. Lu, Y. Cao, C. Xu *et al.*, "Swarm of micro flying robots in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm5954, 2022.

[9] F. Gao, "Px4 - path planning interface," 2022. [Online]. Available: https://www.youtube.com/watch?v=L0fJ0EHHfOA

[10] L. Quan, L. Han, B. Zhou, S. Shen, and F. Gao, "Survey of uav motion planning," *IET Cyber-systems and Robotics*, vol. 2, no. 1, pp. 14–21, 2020.

[11] H. Yu, G. C. E. de Croon, and C. De Wagter, "Avoidbench: A high-fidelity vision-based obstacle avoidance benchmarking suite for multi-rotors," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2023, pp. 9183–9189.

[12] A. D. Householder, G. Wassermann, A. Manion, and C. King, "The cert guide to coordinated vulnerability disclosure," *Software Engineering Institute (Carnegie Mellon University). Disponible en https://bit.ly/3CSCaz5*, 2017.

[13] 2017. [Online]. Available: https://www.first.org/global/sigs/vulnerability-coordination/multiparty/guidelines-v1.0

[14] 2024. [Online]. Available: http://jarus-rpas.org/publications/

[15] Sep 2022. [Online]. Available: https://standards.nasa.gov/standard/NASA/NASA-STD-87398

[16] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "{AFL++}: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[17] T. Kim, C. H. Kim, J. Rhee, F. Fei, Z. Tu, G. Walkup, X. Zhang, X. Deng, and D. Xu, "{RVFuzzer}: Finding input validation bugs in robotic vehicles through {Control-Guided} testing," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 425–442.

[18] R. Han, C. Yang, S. Ma, J. Ma, C. Sun, J. Li, and E. Bertino, "Control parameters considered harmful: Detecting range specification bugs in drone configuration modules via learning-guided search," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 462–473.

[19] H. Kim, M. O. Ozmen, A. Bianchi, Z. B. Celik, and D. Xu, "Pgfuzz: Policy-guided fuzzing for robotic vehicles." in *NDSS*, 2021.

[20] H. Kim, M. O. Ozmen, Z. B. Celik, A. Bianchi, and D. Xu, "Pgpatch: Policy-guided logic bug patching for robotic vehicles," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1826–1844.

[21] H. Choi, S. Kate, Y. Aafer, X. Zhang, and D. Xu, "Cyber-physical inconsistency vulnerability identification for safety checks in robotic vehicles," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 263–278.

[22] H. Kim, M. O. Ozmen, Z. B. Celik, A. Bianchi, and D. Xu, "Patchverif: Discovering faulty patches in robotic vehicles," in *Proceedings of the USENIX Security Symposium*, 2023.

[23] "Px4 open source autopilot," 2022. [Online]. Available: https://px4.io/

[24] Ardupilot, "Ardupilot," 2022. [Online]. Available: https://ardupilot.org/

[25] G. Li, Y. Li, S. Jha, T. Tsai, M. Sullivan, S. K. S. Hari, Z. Kalbarczyk, and R. Iyer, "Av-fuzzer: Finding safety violations in autonomous driving systems," in *2020 IEEE 31st international symposium on software reliability engineering (ISSRE)*. IEEE, 2020, pp. 25–36.

[26] S. Kim, M. Liu, J. J. Rhee, Y. Jeon, Y. Kwon, and C. H. Kim, "Drivefuzz: Discovering autonomous driving bugs through driving quality-guided fuzzing," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1753–1767.

[27] Z. Wan, J. Shen, J. Chuang, X. Xia, J. Garcia, J. Ma, and Q. A. Chen, "Too afraid to drive: systematic discovery of semantic dos vulnerability in autonomous driving planning under physical-world attacks," *arXiv preprint arXiv:2201.04610*, 2022.

[28] J. C. Han and Z. Q. Zhou, "Metamorphic fuzz testing of autonomous vehicles," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 380–385.

[29] Y. Sun, C. M. Poskitt, J. Sun, Y. Chen, and Z. Yang, "Lawbreaker: An approach for specifying traffic laws and fuzzing autonomous vehicles," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[30] H. Tian, Y. Jiang, G. Wu, J. Yan, J. Wei, W. Chen, S. Li, and D. Ye, "Mosat: finding safety violations of autonomous driving systems using multi-objective genetic algorithm," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 94–106.

[31] ApolloAuto, "Github - apolloauto/apollo: An open autonomous driving platform," Dec 2023. [Online]. Available: https://github.com/ApolloAuto/apollo

[32] autowarefoundation, "Github - autowarefoundation/autoware: Autoware - the world's leading open-source software project for autonomous driving," 2015. [Online]. Available: https://github.com/autowarefoundation/autoware

[33] 2023. [Online]. Available: https://www.vector.com/int/en/know-how/virtual-test-drives/openscenario/#

[34] 2021. [Online]. Available: https://avunit.readthedocs.io/en/latest/

[35] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[36] G. Rong, B. H. Shin, H. Tabatabaee, Q. Lu, S. Lemke, M. Možeiko, E. Boise, G. Uhm, M. Gerow, S. Mehta *et al.*, "Lgsvl simulator: A high fidelity simulator for autonomous driving," in *2020 IEEE 23rd International conference on intelligent transportation systems (ITSC)*. IEEE, 2020, pp. 1–6.

[37] X. Zhou, Z. Wang, H. Ye, C. Xu, and F. Gao, "Ego-planner: An esdf-free gradient-based local planner for quadrotors," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 478–485, 2020.

[38] 2023. [Online]. Available: https://docs.px4.io/main/en/flight_controller/autopilot_pixhawk_standard.html

[39] "Raspberry pi," 2022. [Online]. Available: https://www.raspberrypi.com/

[40] "An unprecedented edge ai and robotics platform," 2022. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/

[41] ZJU-FAST-Lab, "Fast-drone-250," 2022. [Online]. Available: https://github.com/ZJU-FAST-Lab/Fast-Drone-250

[42] PX4, "Px4 - path planning interface," 2022. [Online]. Available: https://docs.px4.io/main/en/computer_vision/path_planning_interface.html

[43] ZJU-FAST-Lab, "Ego-planner-swarm," 2022. [Online]. Available: https://github.com/ZJU-FAST-Lab/ego-planner-swarm

[44] Feb 2022. [Online]. Available: https://velodynelidar.com/surround-lidar/

[45] Feb 2024. [Online]. Available: https://www.valeo.com/en/catalogue/cda/long-range-lidar-sensors-scala-gen-3/

[46] Jun 2021. [Online]. Available: https://www.intelrealsense.com/depth-camera-d435i/

[47] 2022. [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/192742/intel-realsense-tracking-camera-t265/specifications.html

[48] D. R. Canelhas, T. Stoyanov, and A. J. Lilienthal, "From feature detection in truncated signed distance fields to sparse stable scene graphs," *IEEE Robotics and Automation Letters*, vol. 1, no. 2, pp. 1148–1155, 2016.

[49] H. Oleynikova, Z. Taylor, M. Fehr, R. Siegwart, and J. Nieto, "Voxblox: Incremental 3d euclidean signed distance fields for on-board mav planning," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 1366–1373.

[50] S. M. LaValle and J. J. Kuffner Jr, "Randomized kinodynamic planning," *The international journal of robotics research*, vol. 20, no. 5, pp. 378–400, 2001.

[51] D. Harabor and A. Grastien, "Online graph pruning for pathfinding on grid maps," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 25, no. 1, 2011, pp. 1114–1119.

[52] A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti, "Libafl: A framework to build modular and reusable fuzzers," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1051–1065.

[53] "Emergency handing in ego-planner," 2023. [Online]. Available: https://github.com/ZJU-FAST-Lab/ego-planner/blob/master/src/planner/plan_manage/src/ego_replan_fsm.cpp#L323

[54] "Emergency handing in ego-planner-swarm," 2023. [Online]. Available: https://github.com/ZJU-FAST-Lab/ego-planner-swarm/blob/master/src/planner/plan_manage/src/ego_replan_fsm.cpp#L593

[55] "Emergency handing in fast-planner," 2023. [Online]. Available: https://github.com/HKUST-Aerial-Robotics/Fast-Planner/blob/master/fast_planner/plan_manage/src/topo_replan_fsm.cpp#L355

[56] "Emergency handing in fuel," 2023. [Online]. Available: https://github.com/HKUST-Aerial-Robotics/FUEL/blob/main/fuel_planner/exploration_manager/src/fast_exploration_fsm.cpp#L335

[57] 2022. [Online]. Available: http://wiki.ros.org/rviz

[58] "so3 quadrotor simulator," 2022. [Online]. Available: https://github.com/ZJU-FAST-Lab/ego-planner/tree/master/src/uav_simulator

[59] X. Zhou, J. Zhu, H. Zhou, C. Xu, and F. Gao, "Ego-swarm: A fully autonomous and decentralized quadrotor swarm system in cluttered environments," in *2021 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2021, pp. 4101–4107.

[60] HT-UAV, "Mao tou ying mini2," 2022. [Online]. Available: http://ht-uav.com/project/25#mini2_infos

[61] AMOVLAB, "P450-nx," 2022. [Online]. Available: https://www.amovlab.com/product/detail?pid=7

[62] "Ego-planner kinodynamic optimization," 2023. [Online]. Available: https://github.com/ZJU-FAST-Lab/ego-planner/blob/0835f963dcfadce156dbec8ab66b93930955a87c/src/planner/plan_manage/src/planner_manager.cpp#L226

[63] "Ego-planner-swarm kinodynamic optimization," 2023. [Online]. Available: https://github.com/ZJU-FAST-Lab/ego-planner-swarm/blob/30de3c9e11e6a2e06d81561f9364f07ceba6774b/src/planner/plan_manage/src/planner_manager.cpp#L228

[64] "Ppfuzzer," 2023. [Online]. Available: https://github.com/ywang-scholar/DPFuzzer/tree/main/RQ1

[65] A. Birk and S. Carpin, "Merging occupancy grid maps from multiple robots," *Proceedings of the IEEE*, vol. 94, no. 7, pp. 1384–1397, 2006.

[66] "Ppfuzzer," 2023. [Online]. Available: https://github.com/ywang-scholar/DPFuzzer/tree/main/RQ2

[67] "Ppfuzzer," 2023. [Online]. Available: https://github.com/ywang-scholar/DPFuzzer/tree/main/RQ3

[68] A. Stocco, B. Pulfer, and P. Tonella, "Mind the gap! a study on the transferability of virtual vs physical-world testing of autonomous driving systems," *IEEE Transactions on Software Engineering*, 2022.

[69] "Ppfuzzer," 2023. [Online]. Available: https://github.com/ywang-scholar/DPFuzzer/tree/main/fixing

[70] "Gazebo robot simulation," 2022. [Online]. Available: https://gazebosim.org/home

[71] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and Service Robotics: Results of the 11th International Conference*. Springer, 2018, pp. 621–635.

[72] S. Kim and T. Kim, "Robofuzz: fuzzing robotic systems over robot operating system (ros) for finding correctness bugs," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 447–458.

[73] M. Klischat and M. Althoff, "Generating critical test scenarios for automated vehicles with evolutionary algorithms," in *2019 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2019, pp. 2352–2358.

[74] A. Li, S. Chen, L. Sun, N. Zheng, M. Tomizuka, and W. Zhan, "Scegene: Bio-inspired traffic scenario generation for autonomous driving testing," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 9, pp. 14 859–14 874, 2021.

[75] X. Zhang, W. Zhao, Y. Sun, J. Sun, Y. Shen, X. Dong, and Z. Yang, "Testing automated driving systems by breaking many laws efficiently," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 942–953.