SWARMFLAWFINDER: Discovering and Exploiting Logic Flaws of Swarm Algorithms

Chijung Jung*, Ali Ahad*, Yuseok Jeon†, and Yonghwi Kwon*
*Department of Computer Science, University of Virginia, Charlottesville, VA, USA
†Department of Computer Science and Engineering, UNIST, Ulsan, South Korea
*{cj5kd, aa5rn, yongkwon}@virginia.edu †ysjeon@unist.ac.kr

2022 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-1316-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/SP46214.2022.9833685

***Abstract*—Inspired by swarms in nature, swarm robotics have been developed to conduct various challenging tasks such as environmental monitoring, disaster recovery, logistics, and even military operations. Despite the significant potential impact of the swarm on society, relatively little attention is given to adversarial scenarios against swarm robotics.**

In this paper, we explore a systematic approach to find logical flaws of the swarm robotics algorithms that adversaries can exploit. Specifically, we develop an automated testing system, SWARMFLAWFINDER, for swarm algorithms. We identify and overcome various challenges in understanding and reasoning about the swarm algorithm execution. In particular, we propose a novel abstraction of robotics behavior, which we call the degree of causal contribution (DCC), based on the idea of counterfactual causality. Then, we build a feedback guided greybox fuzz testing system called SWARMFLAWFINDER, leveraging DCC as a feedback metric. We evaluate SWARMFLAWFINDER with four swarm algorithms conducting navigating, searching, and rescuing missions. SWARMFLAWFINDER discovers 42 logic flaws (and all of them have been acknowledged by the developers) in the swarm algorithms. Our analysis of the flaws reveals that the swarm algorithms have critical logic errors/bugs or suffer from incomplete implementations that can be exploited by adversaries.

I. INTRODUCTION

Swarm robotics revolutionizes how robots can function and what they can accomplish. It has attracted attention for a variety of vital missions, such as search and rescue, that are typically challenging for individual drones to complete. A swarm is more than just a set of drones performing the same operations. Robots in a swarm cooperate with others (e.g., sharing and distributing intelligence) to accomplish tasks.

A swarm operation is controlled by a swarm algorithm, which coordinates the actions of multiple robots. The swarm algorithm's efficacy determines a swarm operation's effectiveness. Logic flaws (i.e., logic bugs or weaknesses) in a swarm algorithm can result in various failures. Consider a swarm searching algorithm that coordinates multiple groups of robots, with robots in the same group sharing information discovered during the mission. The efficiency of the swarm algorithm depends on the number of robots in a group. In such a case, an adversary, who is capable of breaking existing groups into smaller groups, can lead the swarm to undesirable states, significantly slowing down the searching. Such undesirable swarm operations may lead to severe consequences in the wild. For instance, failures in searching/rescuing missions can result in casualties. Failure to search/deliver in military missions can lead to losing a battle. Significantly slowed-down swarm missions in commercial businesses can cause financial loss.

This paper explores a systematic approach for detecting logic flaws in swarm algorithms, particularly in *drone swarms*. Specifically, we develop a greybox fuzz testing technique for swarm robotics, called SWARMFLAWFINDER, that overcomes unique challenges in effectively testing drone swarm algorithms. Given a target swarm algorithm and a swarm mission definition (e.g., the number of drones and mission objectives), SWARMFLAWFINDER introduces attack drones to disrupt the swarm operation. The attack drones aim to interfere with the swarm, attempting to expose logical weaknesses that lead to mission failure, rather than launching naive and overt attacks (e.g., directly crashing into victim drones). A key component in developing SWARMFLAWFINDER is to design an efficient metric that abstracts a given test's effectiveness. Unfortunately, unlike testing traditional software [1]–[3], coverage-based metrics (e.g., basic block, branch/edge, or path coverage) are ineffective in determining a test case's effectiveness and guiding the test generation for swarm robotics because robotics systems are designed to have a relatively less-diverse control flow but significantly more-diverse data variances at runtime.

To this end, a major challenge in SWARMFLAWFINDER is to develop *a metric for the guided fuzzing process*. Inspired by the idea of counterfactual causality, we propose a new metric *the degree of the causal contribution* (or DCC) to abstract the causal impact of attack drones on the target swarm. Specifically, SWARMFLAWFINDER creates multiple perturbed executions (i.e., counterfactual executions) to infer the causality between attack drones and victim drones' behaviors. Based on the inferred causality, we build the DCC to reflect the attack drones' impact on the victim swarm and use DCC to direct the fuzzing process to accelerate the creation of test cases covering unexercised swarm behaviors. We evaluate SWARMFLAWFINDER using four swarm algorithms [4]–[7], finding 42 logic flaws that are all confirmed by the algorithm developers. Our major contributions are summarized as follows:

- We explore the possibility of exploiting swarm algorithms' logic flaws to cause swarm mission failures, solving various technical challenges.
- We propose a concept of *the degree of the causal contribution* (or DCC), based on the idea of counterfactual causality, to abstract the impact of attack drones on a swarm operation.
- We develop a greybox fuzz testing system for drone swarm

DOI 10.1109/SP46214.2022.00084

algorithms called SWARMFLAWFINDER to systematically discover logic flaws in swarm algorithms. It uses DCC as a feedback metric for fuzz testing to mutate the test cases.
- SWARMFLAWFINDER identified *42 previously unknown logic flaws* (all confirmed by the developers) in the four swarm algorithms, and present analysis results including root causes and fixes (34 out of 42 fixes are confirmed).
- We publicly release all the developed tools, data, and results, including SWARMFLAWFINDER, for the community [8].

II. BACKGROUND AND THREAT MODEL

Definition of Swarm Mission and Algorithm. A swarm mission requires the following definitions: (1) the number of drones in a swarm and (2) the objectives of a swarm mission (e.g., the destination or goal). Such definitions can be typically found in configuration files, swarm algorithm's code (i.e., hardcoded), or the algorithms' descriptions (e.g., academic papers or manuals). A swarm algorithm essentially coordinates individual drones to conduct the mission's objectives. In this paper, we consider the swarm algorithms to include logic for both individual drones and the swarm's cooperative behaviors.

Challenges in Testing Swarm Algorithms. A swarm is highly dynamic. During a swarm mission, even a slight impact in one of those inputs (caused by the environment or attack drones) can lead to significantly different swarm behaviors. For instance, assume a moving object is approaching one of the drones in a swarm. The swarm's reaction can be significantly different depending on the approaching angle of the object. Hence, to test swarms effectively, it is desirable to run tests under diverse scenarios to cover various swarm behaviors. However, the swarm's input space (e.g., angles and coordinates of objects) is often too large to cover them exhaustively in practice. To mitigate the large input space, one may try to identify inputs that may exercise a similar swarm behavior (i.e., an equivalent class of the behavior) and prune out those, to improve the testing performance. However, it is challenging to know which inputs exercise a similar swarm behavior.

In typical software testing, coverage-guided fuzzing [9]–[11] solves a similar challenge by using various *code coverage metrics* (e.g., block or edge). It prioritizes the same class of test inputs that have increased the coverage, aiming to exercise diverse program behaviors (i.e., covering diverse execution paths). However, they are not effective in testing robotics systems because their execution is highly iterative. Even with a few tests, majority of the code and branches in robotics systems are quickly covered, while the tests do not cover diverse behaviors. Unlike testing traditional software systems, predicate conditions are not the critical challenges in swarm algorithm testing. Instead, different behaviors are often caused by different values of inputs and internal states of drones.

Greybox Fuzz Testing Approach. SWARMFLAWFINDER chooses to use a greybox fuzz testing approach because other alternatives, whitebox and blackbox approaches, are not as effective as the greybox approach for testing swarm algorithms. Specifically, whitebox approaches [12], [13] often require expensive analyses (e.g., symbolic analysis) on the swarm algorithm. Blackbox approaches [14] do not analyze complex internals of the systems. They rely on correlations between the inputs and observed outputs which are often too coarse grained, to decide the test case mutation strategy.

SWARMFLAWFINDER takes the greybox approach, which monitors an execution (focusing on the poses of drones) to obtain finer-grained information than the blackbox approaches, while not requiring expensive analyses.

Efforts in Dependable Swarm Robotics. There is a line of research on making swarm robotics dependable [15]–[19], where most of them focus on the modeling of swarms, and their discussions are at a high level. Specifically, Winfield et al. [15] define two properties of the swarm systems: liveness (i.e., exhibiting desirable behaviors) and safety (not exhibiting undesirable behaviors such as crashes). They present theoretical models to prove the two properties, leveraging Lyapunov theorems [20]. They also discuss difficulty in testing such as the large input space. Higgins et al. [17] present various security threats to swarm robotics including *intrusion of foreign drones* to a swarm, which is the same threat model of us (i.e., introducing attack drones to disrupt a swarm). Sargeant and Tomlinson [16] present models of malicious swarms aiming to make a victim swarm operation inefficient.

Compared to the above work [15]–[17], we aim to identify *concrete logical flaws from real algorithms* via testing. In the context of [15], SWARMFLAWFINDER can find flaws delaying mission completion and crashing drones in a swarm that can be considered 'liveness' and 'safety' violations, respectively. To the best of our knowledge, SWARMFLAWFINDER advances state-of-the-art swarm testing, especially in testing efficiency and quality, mitigating the incompleteness of the testing discussed in [15]. Note that while [17] presents malicious swarm models, their models are not concrete. For example, they describe high-level classes of threats such as 'mobility' and 'controllability' issues. Instead, we find concrete logic flaws with root causes. In other words, while some logic flaws we find can relate to [17]'s definitions (In Table III, C1-5 and C2-4 can be classified as mobility and controllability issues, respectively), *all the logic flaws we find are previously unknown*, meaning that they are *newly discovered*. Similarly, [16] presents an example swarm threat scenario called landmine, which has a similar objective (i.e., conducting a search) to two swarm algorithms we evaluate (A2 and A3). We also find logic flaws that slow down a swarm's progress (See C2-3, C2-4, C3-1, and C3-2 in Table III). However, [16]'s discussions are conceptual and all the discovered flaws we find are new. Note that the models in [15]–[17] can be used to define additional mission failure criteria for our testing.

Besides, there are groups of researchers conducting in-depth analysis in designing and modeling swarm algorithms. Taylor et al. [18] discuss the effectiveness of adding collision avoidance algorithms to existing swarm algorithms. It concludes that it is recommended to design swarm algorithms with collision avoidance in mind, rather than adding the collision avoidance algorithm later. In our paper, all the four evaluated algorithms are designed with collision avoidance in mind (i.e., we do

not observe a clear separation of the collision avoidance logic from swarm algorithms). Hamann et al. [19] model swarm robotics using statistical physics, showing that their models are effective. Our work focuses on finding concrete logic flaws in a concrete implementation of an algorithm, which is difficult to achieve with the modeling approach.

Threat Model. We assume an adversary knows the target swarm mission and its swarm algorithm and can launch *external* attack drones to thwart the target swarm operation. However, the adversary does not have access to the target drone's device, hence cannot compromise the drone's software/hardware. The adversary prefers subtle attacks that do not make physical contact (e.g., crashing into the victim drones) due to its economic benefit and subtleness. Note that a naive crashing attack is not practical and scalable for a large-scale swarm mission since crashed attack drones are not reusable by the adversary, limiting the attack capability.

We target autonomous swarm algorithms and do not target human-controlled swarms. If a swarm is a mixture of human and autonomous control, we target the part of the swarm with autonomous control. In practice, autonomous control is required in many cases, such as conducting a long-distance mission covering areas without communication infrastructure (e.g., military mission) or a large-scale swarm mission (e.g., a search/rescue mission over a large area). Our focus is to find logic flaws in swarm algorithms. Traditional software/hardware vulnerabilities of drones such as GPS jamming/spoofing [21], [22] and network packet injections are not our focus.

III. Motivating Example

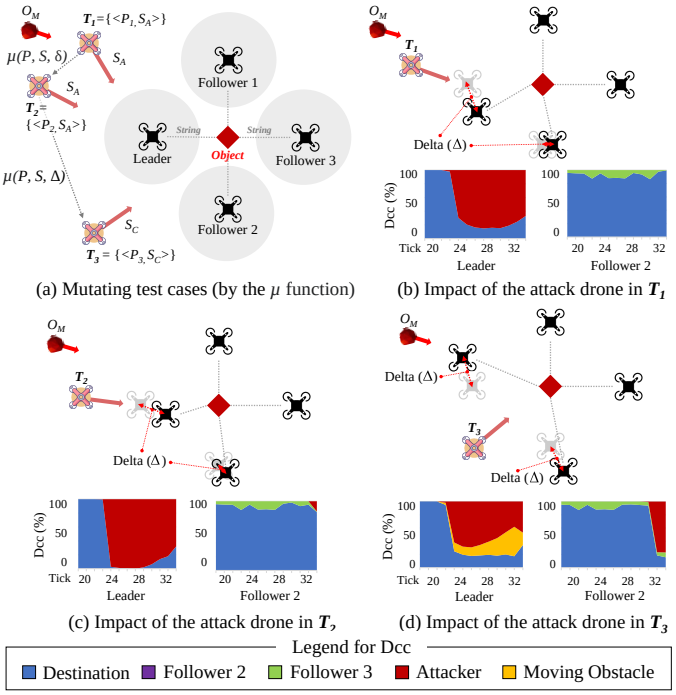
We use a drone swarm mission running Adaptive Swarm [4] to show how SwarmFlawFinder discovers logic flaws.

Target Swarm Mission. The target swarm aims to deliver an object that requires four drones' cooperation as shown in Fig. 1-(a). Each drone is attached with a string to hold the object. Typically, it takes 189.4 (±5.8) ticks to complete (We profile 100 runs of the mission to obtain the completion time).

Adversary. We assume an adversary wants to discover the swarm algorithm's logic flaws that can be exploited by an attacker controlled external drone, in order to fail the mission. We consider the swarm mission is *failed* if the swarm does not reach the destination in 400 ticks (i.e., two times longer than the typical mission completion time mentioned above).

Logic Flaw Discovery. SwarmFlawFinder conducts guided fuzz testing via the following four steps.

1) Test Creation: Given the target swarm mission, we create the initial test (T_1 in Fig. 1-(a)). A test case consists of two elements: the attack drone's pose (or location; P) and an attack strategy (S). For the initial test, we randomly pick P and S where P being near a victim drone while avoiding being too close to the victim drone (i.e., indicated by the gray area in Fig. 1-(a)), because choosing such a value may cause a crash immediately after the spawn. The attack strategy S represents how the attack drone will act after the spawn. There are four strategies S_1~S_4: S_1 pushes a victim drone against its flight



(a) Mutating test cases (by the μ function) (b) Impact of the attack drone in T_1

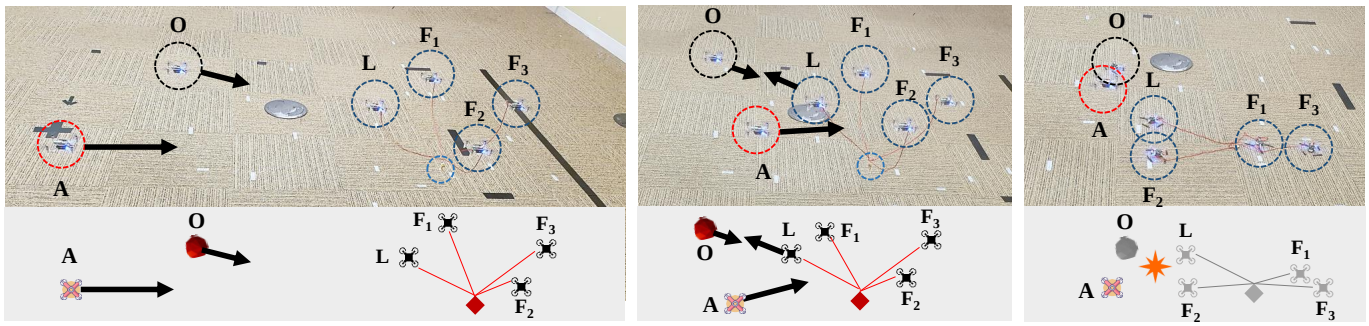
(c) Impact of the attack drone in T_2 (d) Impact of the attack drone in T_3

Fig. 1. SwarmFlawFinder in action on the motivation example.

direction and S_3 represents a strategy that moves between two victim drones. Other strategies can be found in § IV-A.

2) Test Evaluation and* Dcc *Computation: We run the test T_i and measure the attack drones' impact on the victim swarm. We propose the concept of the degree of causal contribution (or Dcc), which is based on the principle of counterfactual causality [23], [24], to measure the impact. Briefly, a causal relationship between an attack drone and a victim drone is inferred by comparing an execution with the attack drone and its *counterfactual* execution, which does not include the attack drone. Any observable differences between the two executions essentially represent the causality between the attack and victim drones (Details about the counterfactual causality are in § IV-B). Specifically, for each victim drone, we identify all external objects that can affect the swarm operation. In this example, the external objects for a victim drone (e.g., Leader) include an attack drone, three victim swarm's drones (Follower 1~3), and a moving object (O_M). To compute Dcc, for every external object, we run an additional test without the external object. Any observed differences on the victim drone's pose between the tests with and without the object (e.g., represented as Δ in Fig. 1) are collected. We repeat this for all external objects, and accumulate the Δ values to get the Dcc values, shown at the bottom of Fig. 1-(b)~(d).

3) Test Mutation Guided by* Dcc*: After each test's execution, SwarmFlawFinder checks whether there was a previous test that has a similar Dcc of the current test. If there are no similar Dcc values observed previously, we consider that the current test exercises a new behavior of the target swarm. Hence, SwarmFlawFinder tries to prioritize tests that are similar to the current test. It derives the next test by mutating the test case slightly, denoted by $\mu(P, S, \delta)$. Observe



(a) Attack drone and obstacle approach the victim swarm (b) Attack drone influences a victim drone (c) Leader drone crashed into the obstacle

Fig. 2. Physical experiment reproducing the crash shown in Fig. 3 (L means Leader and $F_{1\sim3}$ indicates Follower 1~3).

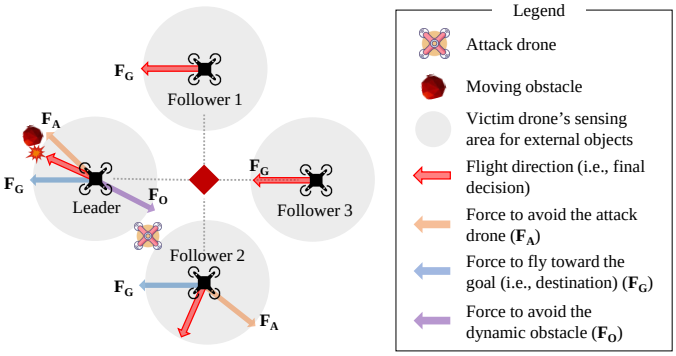


Fig. 3. Crash (caused by a logic flaw) found by SwarmFlawFinder.

that T_1 and T_2 in Fig. 1 have the same S_1 (i.e., not mutated). If the current test's Dcc is similar to one of the previously observed Dcc values (e.g., Dcc of Fig. 1-(b) and (c) are similar), SwarmFlawFinder mutates the current test more significantly to derive a completely new test case for the next test (e.g., T_3 is derived by mutating both P and S of T_2).

4) Repeating Test Execution and Mutation: We repeat the Step 2 and Step 3 for a given amount of time (i.e., timeout): 24 hours in this example. During the testing process, we observe a test case execution leading to a swarm mission failure due to a crash between a victim drone and the moving obstacle (O_M). Note that O_M is not an attacker controlled object. The victim swarm is capable of avoiding O_M without an attack drone introduced by our system. SwarmFlawFinder also logs the details of the test causing mission failures (e.g., attack drone's pose and strategy) for analysis.

Logic Flaw in the Algorithm. Fig. 3 explains the details of a logic flaw discovered by SwarmFlawFinder. In this scenario, three forces are considered to determine the final flight direction of the victim drones. First, all four victim drones try to move toward the goal, denoted by F_G. If there are no other forces to consider, F_G becomes the final flight direction denoted by the red arrow. Follower 1 and 3 are such cases. Second, when an attack drone comes close to a victim drone (e.g., Leader and Follower 2 in Fig. 3), the victim drone tries to avoid it, causing F_A. Third, when a moving obstacle approaches the victim drone, it tries to avoid the obstacle (F_O). Note that when multiple forces are involved, the final flight direction is determined by adding all the forces' vectors. In this example, when the attack drone flies in the middle of Leader and Follower 2, the sum of F_G, F_O, and F_A of Leader makes the drone move towards the obstacle, leading to the crash.

Physical Experiment. To show that the identified logic flaw can be exploited in the real world, we reproduce the motivation example with real drones in our lab environment, as shown in Fig. 2. Observe that we present the photos of real drones on the upperside along with the simplified versions of the photos on the bottom. A and O represent the attack drone and the moving obstacle, respectively. The victim drones are connected with red strings to hold an object (illustrated as a red diamond on the bottom). Fig. 2 shows three steps: (a) the attack drone and obstacle are approaching the victim swarm. (b) the attack drone influences a victim drone's decision, making it move toward the obstacle. (c) the obstacle and the victim drone influenced by the attack drone crashed, resulting in the entire swarm crashed onto the ground (illustrated by the gray color).

Generality. We further analyze the crash in detail and discover that Adaptive Swarm [4] does not handle multiple obstacles well in general, meaning that the above crash is not an accidental crash but it is caused by a fundamental weakness of the algorithm. Details of the root cause of this error are presented in § V-B (*C1-2. Naive multi-force handling*).

IV. Design

Fig. 4 shows the overview of SwarmFlawFinder. It takes a target swarm algorithm and a swarm mission (including the definition of mission success and failure) as input. It runs an initial test with attack drones (§ IV-A). If a test mission finishes successfully (❶), it conducts execution perturbation (§ IV-B) to understand whether the current test exercised a new behavior of the swarm or not. Based on the result, SwarmFlawFinder mutates the current test and continues testing (❷, § IV-C). If a test leads to a mission failure (❸), the attack drones' configuration is obtained as output (❹). It repeats the above process until it reaches a predefined timeout.

A. Test-run Definition and Creation

A test-run is defined as a set of tuples $< P, S >$ where P and S represent an attack drone's pose and its strategy respectively. A test with n attack drones is composed of multiple tuples: $\{< P_1, S_1 >, < P_2, S_2 >, ..., < P_n, S_n >\}$. To facilitate the discussion, we first focus on testing with a single attack drone. We discuss testing with multiple attack drones in § IV-D.

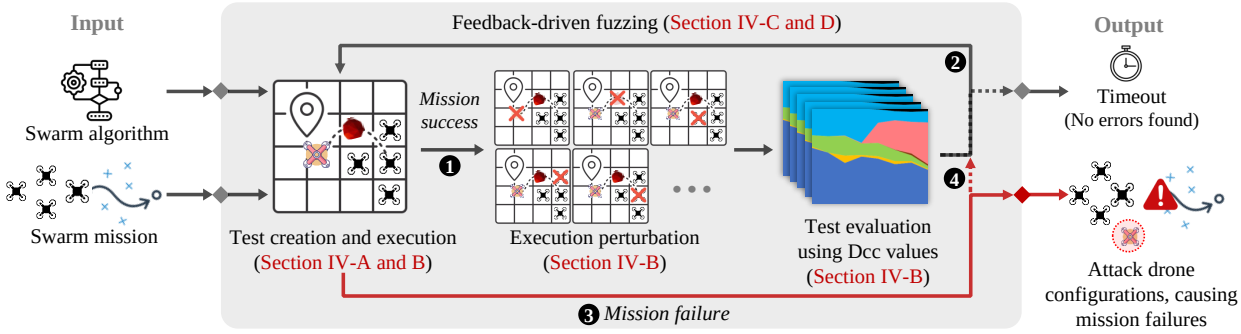


Fig. 4. Overview of SWARMFLAWFINDER. (The shaded area represents SWARMFLAWFINDER with input and output on the left and right respectively)

Attack Drone's Pose (P). P represents the initial pose of the attack drone in a test. P is essentially a point in 3D space in drone swarms, represented by three values on xy, xz, and yz-planes: $< x, y, z >$. P's value range is large as it can be any point in 3D space except for the points that are close to victim drones (which can cause crashes even before a victim drone tries to avoid collisions). For example, if a victim drone's sensing area (i.e., the area that the victim drone can detect an object) is defined as $x \times y \times z$ (length × width × height), we only allow a value of P that is outside of the $x \times y \times z$ from the center of each victim drone. The sensing area can be obtained by running a simple test with an external object and observing the distance the victim drone starts to avoid the object. After the attack drone is spawned at P, it moves toward the victim drone to execute its attack strategy S (explained in the next paragraph). Different P values can lead to different timings of the attack drone approaching the victim swarm.

Attack Strategy (S). After an attack drone is spawned at P, it detects the victim swarm and moves near the swarm. Then, it conducts an attack based on the strategy S defined as follows (An illustration of the strategies can be found in [8]).

1. *Pushing Back (S_1):* An attack drone tries to push back a victim drone (i.e., against the victim drone's flight direction). In a swarm, this strategy typically delays the progress of the swarm reaching the destination.
2. *Chasing (S_2):* An attack drone closely follows a single victim drone in a swarm. It typically causes a victim to speed up, often making it difficult for the victim to control itself from crashing into other objects.
3. *Dividing (S_3):* An attack drone flies into the middle of two victim drones to divide a group of drones into smaller groups. It aims to disrupt the swarm's collective operation, making the swarm sparse or smaller sized.
4. *Herding (S_4):* It aims to change the direction of an entire swarm or the size of the swarm via attack drones pushing victim drones from the outmost layer of the swarm.

B. Test Execution and Evaluation

Initial Test Creation and Execution. We create the initial test case by randomly choosing P and S for a single attack drone. We run the created test case which spawns an attack drone at P with an attack strategy S.

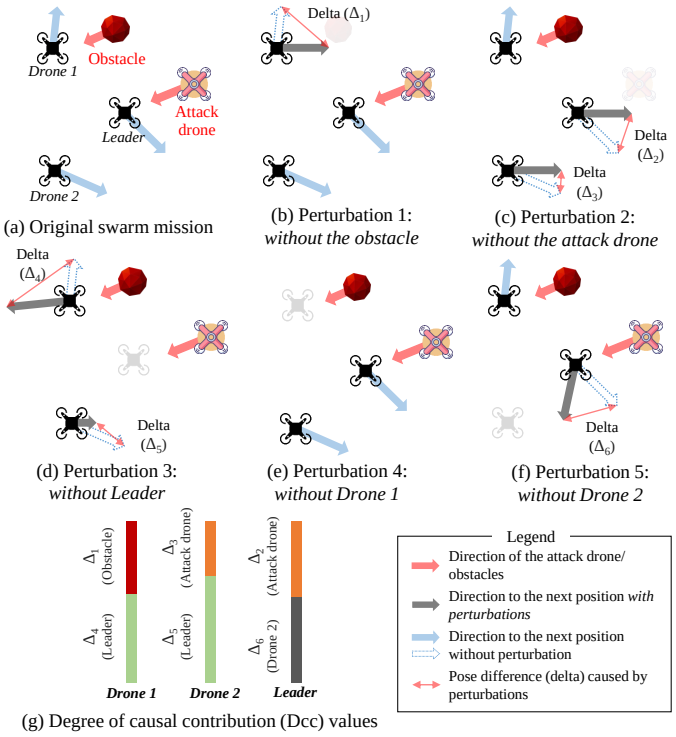
Test Evaluation. After a test, we evaluate the effectiveness of the test. If the test case (i.e., $< P, S >$) effectively exercises a new behavior of the victim swarm, we consider the test case is effective and try to run similar tests with a slight mutation (e.g., changing P to have less than 1 meter change from the original P and do not change S). Otherwise, we try to mutate the current test case significantly to derive a completely different test that may exercise a new behavior of the victim swarm. For example, we consider a significant mutation to be (1) mutating P to have more than 1 meters (10 times of the attack drone's size) change and (2) selecting a different S.

• *Challenges:* Unfortunately, coverage based metrics (e.g., instruction or branch coverage) that are commonly used in traditional software testing do not work well for swarm algorithms because the algorithms are highly iterative. We observe that even between significantly different tests, the coverage metrics stay similar. Alternatively, one may record victim drones' poses (e.g., coordinate values) during the test run and use the pose trace. However, the pose trace is too sensitive, meaning that even for very similar tests, they may differ significantly. Even running the same test multiple times likely results in different poses, due to the non-deterministic nature of swarm robotics. Hence, pose traces are not desirable.

• *Our solution:* We focus on *the attack drones' impact on the victim swarm*, where the impact can be intuitively measured by *the victim drones' reactions* to the attack drones. To quantify the impact (or swarm's reactions), we propose *the degree of the causal contribution* (or DCC). The idea behind the DCC is counterfactual causality [23], [24] which explains the meaning of causal claims in terms of counterfactual conditionals: "If X had not occurred, Y would not have occurred."

> **Counterfactual Causality** [23] is the most widely used definition of causality. We adapt the above counterfactual conditional statement to the context of inferring adversaries' impact on drone swarm algorithms' execution. Specifically, a victim drone's behavior B is causally dependent on an adversary A, if A did not exist, B would not exist. To this end, we conduct additional experiments to infer the causality between A and B.
>
> Given an execution E_{org} of a swarm algorithm, we create a new counterfactual execution E_{cf} that does not include A, to test the counterfactual condition. From the above definition, we can infer the causality between A and B as follows. If B is only observed in E_{org} but not in E_{cf}, B is causally dependent on A. Note that B in our context is not a binary

but a difference (i.e., delta) between the two executions. In other words, we aim to infer the causal relationship between A and B where B is the behavior difference of a victim drone between E_{org} but not in E_{cf}.



(a) Original swarm mission
(b) Perturbation 1: *without the obstacle*
(c) Perturbation 2: *without the attack drone*
(d) Perturbation 3: *without Leader*
(e) Perturbation 4: *without Drone 1*
(f) Perturbation 5: *without Drone 2*
(g) Degree of causal contribution (Dcc) values

Fig. 5. Dcc computation via perturbed swarm executions.

We compute Dcc values by (1) perturbing the original swarm mission's execution, (2) comparing the original swarm mission with the perturbed swarm mission executions, and (3) aggregating the differences of each victim drone in the swarm.

SwarmFlawFinder perturbs all objects including victim drones, objects, and attack drones, one by one in each perturbed execution. Fig. 5-(a) shows the original swarm mission including 3 victim drones, 1 moving obstacle, and 1 attack drone. SwarmFlawFinder creates 5 perturbed executions.

1. *Removing the obstacle (b):* The obstacle is removed from the swarm mission. Observe that Drone 1 is now flying toward the east (gray arrow). The difference between the original swarm mission is identified and annotated by Δ_1.
2. *Removing the attack drone (c):* Without the attack drone, two victim drones (Leader and Drone 2) move toward the east (i.e., the original destination) as annotated by Δ_2. Drone 2's flight direction is also changed (Δ_3) because, in the original execution (a), it flies slightly south to avoid the Leader drone that is affected by the attack drone.
3. *Removing the Leader (d):* In this swarm algorithm, non-leader drones are instructed to follow the leader drone, which aims to fly toward the destination. Without the leader, the other drones do not try to fly toward the east. Drone 1 reacts to the obstacle more actively since it does not need to care about the destination (Δ_4). Drone 2 also slows down and does not need to follow the leader (Δ_5).
4. *Removing the Drone 1 (e):* Drone 1 does not affect other victim drones' (Leader and Drone 2) behaviors. We observe no delta values in this experiment.
5. *Removing the Drone 2 (f):* Without Drone 2, the leader drone tends to fly toward the west more to avoid the obstacle (Δ_6). In the original swarm mission, the leader flies toward the south-east to avoid Drone 2.

Fig. 5-(g) shows Dcc values computed from the perturbed executions at the moment illustrated in Fig. 5. For each victim drone, it is the percentage of aggregated Δ values. Note that we collect Dcc values throughout the entire swarm mission.

Algorithm for Dcc Computation. `SwarmDcc()` in Algorithm 1 shows the algorithm to compute Dcc values. Specif-

Algorithm 1: Feedback based fuzz testing

Input : D_v: a set of variables representing victim drones.
D_a: a set of variables representing attack drones.
O_w: a set of variables representing objects in the world.
$T_{timeout}$: the maximum time limit for the testing (i.e., timeout).
Output: E_{failed}: a set of executions that were failed due to the attack drones.

```
1  procedure SwarmDcc(E, D_v, D_a, O_w, T_end)
2      t ← 0
3      while t ≠ T_end do
           // Each victim drone d
4          for d ∈ D_v do
5              ΔTotal ← 0
6              O_all ← D_v ∪ D_a ∪ O_w
7              P_org ← GetPose (E, d, O_all, t) // Pose of a victim drone d at t
               // Each variable o representing objects including attack/victim
                  drone and obstacles
8              for o_i ∈ O_all do
9                  o_bak ← o_i // Save o_i
10                 o_i ← ∅ // Removing an object o_i
11                 P_i ← GetPose (E, d, O_all, t) // Pose of d at t without i
12                 Δi ← | P_org − P_i | // Δ for o_i via Euclidean Distance
13                 ΔTotal ← ΔTotal + Δi
14                 o_i ← o_bak // Restore o_i
15             for o_i ∈ O_all do
16                 Dcc(d, t) ← Dcc(d, t) ∪ < o_i, (Δi / ΔTotal) >
17         t ← t + 1
18     return Dcc
19 procedure FuzzTesting(D_v, D_a, O_w, T_timeout)
20     E_failed ← {}
21     E_cur ← CreateInitialTest(D_v, D_a, O_w) // Create the first test
22     N_threshold ← 0.87 // NCC threshold (configurable).
23     while the elapsed time of testing did not reach T_timeout do
           // Run a test with the current configuration. If the current victim
              mission fails, add the execution to the output.
24         if RunSwarm(E_cur) = MISSION_FAILURE then
25             E_failed ← E_failed ∪ E_cur
           // Obtain Dcc values for the current test
26         Dcc_cur ← SwarmDcc(E_cur, D_v, D_a, O_w, time(E_cur))
           // Check whether the current test produce Dcc values different enough
27         IsNewDcc ← True
28         for r ∈ D_v do
29             for d_i ∈ Dcc_prev(r) do
30                 if GetNCC(d_i, Dcc_cur(r)) > N_threshold then
31                     IsNewDcc ← False
32                     break
33         Dcc_prev ← Dcc_prev ∪ Dcc_cur
           // The test did not find the obtained Dcc values are different enough,
              meaning that it is similar to one of the previous tests
34         if IsNewDcc = False then
35             E_cur ← MutateTest(E_cur, ℝ) // Mutate the test significantly
36         else
37             E_cur ← MutateTest(E_cur, δ) // Mutate slightly (δ)
38     return E_failed
```

ically, DCC values are computed for each victim drone specified by D_v. The `for` loop from line 4 to line 16 describes the DCC computation for each drone. SWARMFLAWFINDER runs multiple tests with perturbations that remove one of the attack drones (D_a), obstacles (O_w), and the victim drones (D_v) specified as input (Lines 8~14). In particular, P_i (line 11) and P_{org} (line 7) represent the pose of a drone with and without the perturbation. We then compute the euclidean distance between the two trajectories (Δi at line 12, which is essentially Δ in Fig. 5). To understand the attack's impact on the entire swarm, we compute all the delta values for all victim drones (see the nested for loops at lines 4~16 and 8~14).

DCC is computed by adding all the delta values computed (line 13) and then calculating each delta value's proportion in the total accumulated delta value (in percentage) (lines 15~16).

C. DCC Guided Fuzz Testing

Abstracting Swarm Missions via DCC. Fig. 6 shows a series of DCC values computed throughout the swarm mission (0~180 ticks). X-axis and Y-axis represent the time and stacked Δ values, respectively. Intuitively, we use the series of DCC values to represent a swarm mission. When we identify two tests that have a similar series of DCC values, we consider them similar. To compare two number series, we leverage NCC (Normalized Cross Correlation) [25] which is commonly used to compute the similarity between data in various fields [26]–[28]. Fig. 6 shows examples of NCC scores from different DCC values: (a) shows Dcc values from the original execution. (b) and (c) are DCC values from two different runs. Note that when DCC values from two executions have different lengths (i.e., running time), we scale one of the execution's DCC values to another execution (i.e., interpolation), then compute the NCC. However, if two executions' running times are different more than two times, we consider they are different.

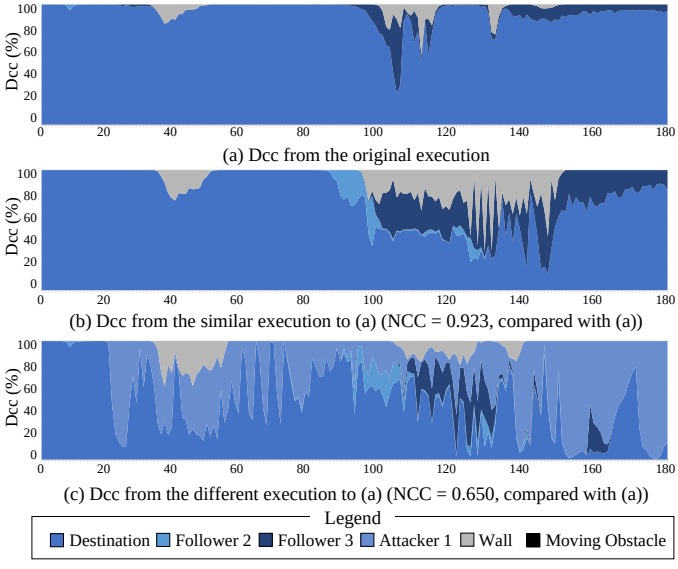


Fig. 6. Example of NCC scores from three executions.

Using NCC [25] of DCC Values to Guide Testing. After every test, we store observed DCC values from the test. Then, we determine whether the current test exercises new behaviors of the swarm by computing NCC scores with the previously observed (and stored) DCC values (lines 26~32 in Algorithm 1). Specifically, after each run, for each victim drone, we compute NCC scores against all of the previously observed DCC values for the victim drone. If there is a previous test run with an NCC score larger than a threshold (0.75~0.87 in this paper, line 22 in Algorithm 1), we consider that the current test run is similar to the compared run, meaning that we consider it *did not exercise* a substantially new swarm behavior. Hence, we aim to mutate the test (i.e., the pose and strategy) significantly to derive the next test (line 35, $\mathbb{R}$ representing a large random value). If there is no previous test case with an NCC score smaller than the threshold, it means that the current test has DCC values that *have not been seen* yet. Then, we mutate the test slightly to derive a new test since we may find new swarm behaviors from a test similar to the current test (line 37, δ representing a small random value).

Note that the NCC threshold value is configurable, and it does not affect the validity of the testing. If the threshold is ill-configured, SWARMFLAWFINDER may finish the testing process early (if the value is too high) because significantly different test runs will be considered similar. If the configured value is too low, the testing will take longer as it considers more tests are different. To find a proper NCC threshold, we run 100 runs for the same initial test of a given swarm mission (without any changes), and then take the lowest value of the measured NCC scores as shown in Table II.

Algorithm. `FuzzTesting()` in Algorithm 1 describes the entire fuzz testing process of SWARMFLAWFINDER including measuring the swarms' behaviors against attacks and mutating tests based on the measured impacts. The algorithm takes four inputs: (1) D_v: a configuration of the victim drones, including their poses and goals, (2) D_a: a configuration of attack drones consisting of attack drones' poses and strategies, (3) O_w: objects such as walls and moving obstacles that affect the victim and attack drones during the mission, (4) $T_{timeout}$: the time limit for the entire testing process. Typically, this is set for longer than several hours (e.g., 24 hours). The output (i.e., return) is E_{failed} which is a set of executions where the missions were failed due to the conducted attacks (line 38).

D. Testing with Multiple Attack Drones

Algorithms that run significantly distributed drone swarms may require SWARMFLAWFINDER to test with multiple attack drones. For example, for a swarm algorithm that maintains a number of small swarm groups spread over a large area, a single attack drone may only affect one of the groups, making it difficult to find a logic flaw. To handle this, SWARMFLAWFINDER automatically adds an additional attack drone and repeats the testing if the entire testing process failed to find attacks. Note that adding N additional attack drones causes roughly $5*N\%$ overhead on average (for all the algorithms we evaluated). Details of the number of additional attack drones and additional overhead are presented in [8].

Mutating Tests with Multiple Attack Drones. As described in § IV-A, a test run with multiple drones is defined as a test

case with multiple tuples such as $\{< P_1, S_1 >, < P_2, S_2 >, ..., < P_n, S_n >\}$, where each tuple represents an attack drone. When there are multiple attack drones in a test, we may observe the changes of DCC values caused by multiple attack drones. It is critical to identify which attack drone is effective in exercising a new behavior of the swarm to choose the mutation strategy (i.e., mutating significantly or slightly as shown in lines 35 and 37 of Algorithm 1). We apply the mutation for each attack drone (i.e., each tuple) so that DCC value changes caused by an attack drone would not mutate the other attack drones.

For each attack drone, SWARMFLAWFINDER identifies all the victim drones' DCC values that are affected by the attack drone. There are two cases of victim drones affected by an attack drone: *directly* and *indirectly*. First, the victim drone is *directly* affected when we observe the attack drone's delta value in the victim drone's DCC values. Second, the victim drone is *indirectly* affected by the other victim drone that is directly affected by the attack drone (i.e., a cascading effect). To this end, we check the DCC values of the victim drones to identify the drones affected by each attack drone and compute the NCC values for the identified victim drones. We present an example scenario with multiple attack drones on [8].

V. EVALUATION

A. Experiment Setup

1) Selection of Target Swarm Algorithms: We search open-sourced research projects related to swarm robotics for the last ten years, from 2010 to 2021. We listed 44 academic papers and 29 public GitHub repositories from the initial search. From the 44 papers, 17 of them provide source code, resulting in 46 available algorithms. However, 20 out of 46 algorithms are not executable (e.g., the source code is incomplete and not compilable) or partially implemented (e.g., only implementing algorithm logic), leading to 26 runnable algorithms. Finally, we prune out 22 out of 26 algorithms since they do not exhibit *collective (or cooperative) behaviors* or allow external objects such as our attack drones (hence cannot implement our approach). Specifically, swarm algorithms that are a collection of individual drones lacking cooperative interactions between the neighbor drones [29]–[38] are not considered.[1] To this end, we choose four runnable algorithms that exhibit collective swarm behaviors and allow us to introduce external objects. Details of the selection process can be found in § IX-A.

TABLE I
SELECTED SWARM ALGORITHMS FOR EVALUATION

ID	Name	SLOC	Language	Algorithm's Objective
A1	Adaptive Swarm [4]	3,091	Python	Multi-agent navigation
A2	SocraticSwarm [5]	9,920	C#	Coordinated search
A3	Sciadro [6]	3,851	Netlogo	Distributed target search
A4	Pietro's [7]	752	Matlab	Coordinated search and rescue

Selected Target Algorithms. Table I presents the selected four swarm algorithms and Fig. 7 shows visualizations of the swarm algorithms using the Gazebo simulator [39].

[1]If a drone in an algorithm does not recognize other drones as cooperating units (e.g., other drones are considered as obstacles), we exclude the algorithm.

A1. Adaptive Swarm [4] aims to move a swarm of (up to 20) drones, from the current position to a predefined destination (shown as a yellow path in Fig. 7-(a)) while maintaining a formation and avoiding obstacles.

A2. SocraticSwarm [5] conducts a swarm searching mission, where individual drones actively interact with neighbor drones to share information, as shown in Fig. 7-(b).

A3. Sciadro [6] runs multiple swarms to search targets distributed over a wide range of areas, as shown in Fig. 7-(c). Swarm groups can be dynamically changing at runtime, allowing individual drones joining and leaving a swarm.

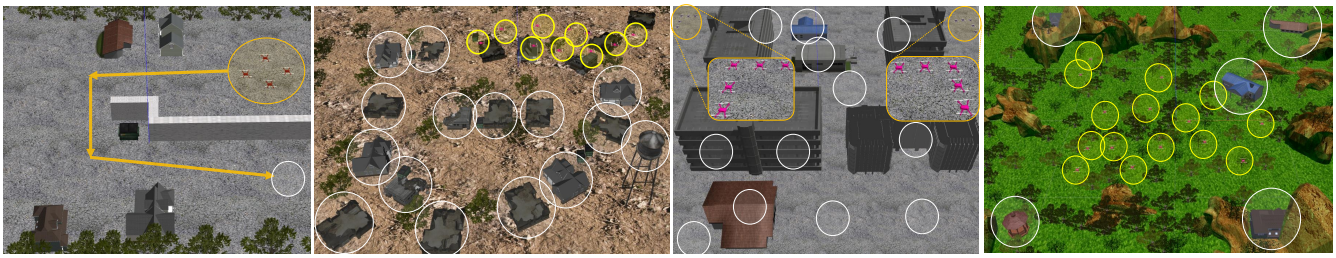
A4. Pietro's algorithm [7] aims to achieve a cooperative rescue mission. Fig. 7-(d) shows an example mission: searching and rescuing targets inside various structures. The process is accelerated with more participating drones.

TABLE II
FUZZ TESTING CONFIGURATIONS

ID	Completion time (sec)	200% Deadline	NCC threshold	Mutation (δ / $\mathbb{R}$)	# of victim drones	Time for testing
A1	189.4	400	0.87	0.4 / 0.8	4	24 hrs
A2	90.11	200	0.82	50 / 100	8	24 hrs
A3	1,756.13	3,500	0.85	25 / 50	10	24 hrs
A4	715.41	1,400	0.75	10 / 5	15	24 hrs

2) Experimental Configurations: Table II shows how we define mission failures in the four selected swarm algorithms' missions. We consider a swarm mission failed (1) if it takes longer than two times of its typical mission completion time to accomplish its given goals or (2) a drone in the swarm crashes into an object or another victim drone. Note that we do not try opportunistic attacks such as blocking the target point to prevent the mission completion. Similarly, we do not count attack drones crashing into the victim drone as a failure. Our attack drones are designed not to crash into victim drones directly. The third column defines the 200% deadline, which is essentially the time we consider a mission fails if it exceeds. They are roughly more than 200% of the completion times. The fourth column shows the NCC threshold used in the experiments for each algorithm. To get the typical mission completion time and NCC threshold for each algorithm, we run each mission 100 times and get an average completion time without any interventions (i.e., without attack drones). We also find the NCC thresholds by taking the lowest NCC values from the 100 test runs. The fifth column shows the distance values used to apply slight (δ) and significant ($\mathbb{R}$) mutation in each algorithm. The sixth column shows the number of victim drones for each algorithm, varying from 4 to 15 drones. Finally, the last column presents that we run SWARMFLAWFINDER on each algorithm for 24 hours.

3) Implementation and Setup: We implement prototypes of SWARMFLAWFINDER for each algorithm in the programming language that the original algorithm is written in: Python, C#, Netlogo, and Matlab. Our implementation includes modifications of existing simulators/emulators. To this end, we write 839, 331, 422, and 230 SLOC for implementing SWARMFLAWFINDER for A1~A4, respectively. Our analysis tool for NCC and the map of A3 is written in R (820 lines).



(a) Adaptive Swarm (Navigation) (b) SocraticSwarm (Coordinated search) (c) Sciadro (Distributed search) (d) Pietro's (Search and rescue)

Fig. 7. Visualizations of the selected algorithms' missions. Yellow and white circles indicate swarm drones and search/rescue targets or the destination.

TABLE III
FUZZ TESTING RESULTS

ID	Mission Failure and Root Cause	# of Exec.	Uniq.	Confm.
A1	**Crash between victim drones**	**273**	**9**	
	– C1-1: Missing collision detection	86	4	✔
	– C1-2: Naive multi-force handling	176	4	✔
	– C1-3: Unsupported static movement	11	1	✔
	Crash into external objects	**435**	**8**	
	– C1-1: Missing collision detection	88	3	✔
	– C1-2: Naive multi-force handling	326	3	✔
	– C1-3: Unsupported static movement	3	1	✔
	– C1-4: Excessive force in APF	18	1	✔
	Suspended progress	**671**	**2**	
	– C1-5: Naive swarm's pose measurement	242	1	✔
	– C1-6: Insensitive object detection	429	1	✔
	Slow progress	**175**	**1**	
	– C1-6: Insensitive object detection	175	1	✔
	Total	1,554/1,724	**20**	
A2	**Crash between victim drones**	**28**	**3**	
	– C2-1: Overly-sensitive object detection	28	3	✔
	Suspended progress	**119**	**1**	
	– C2-2: Indefinite wait for crashed drones	119	1	✔
	Slow Progress	**608**	**4**	
	– C2-3: Long deadline for assigned task	586	3	✔
	– C2-4: Drones detaching from a swarm	22	1	✔
	Total	755/990	**8**	
A3	**Crash into external objects**	**47**	**2**	
	– C3-1: Naive/faulty detouring method	10	1	✔
	– C3-2: Insensitive object detection	37	1	✔
	Slow progress	**240**	**4**	
	– C3-1: Naive/faulty detouring method	23	2	✔
	– C3-2: Insensitive object detection	217	2	✔
	Total	287/811	**6**	
A4	**Crash between victim drones**	**230**	**3**	
	– C4-1: Naive detouring method	216	1	✔
	– C4-2: Detouring without sensing	14	2	✔
	Crash into external objects	**630**	**3**	
	– C4-1: Naive detouring method	599	1	✔
	– C4-2: Detouring without sensing	31	2	✔
	Slow progress	**1,228**	**2**	
	– C4-3: Insensitive object detection	1,228	2	✔
	Total	2,088/2,469	**8**	

Environment Setup. For our evaluation, we use a machine with i7-9700k 3.6Ghz and 16GB RAM, running 64-bit Linux Ubuntu 16.04 (for A1, A3, and A4) and Windows 10 (for A2).

B. Effectiveness in Finding Logic Flaws

Table III presents the number of executions exhibiting logic flaws identified by SWARMFLAWFINDER for each algorithm. In total, we find 4,684 executions leading to mission failures for the four algorithms: 1,554 from A1, 755 from A2, 287 from A3, and 2,088 from A4 (in the third column). After pruning out similar executions, we find 42 distinct mission failures, that are attributed to 15 different root causes (C1-1~C4-3)[2]. The unique number of failures are presented in the fourth column and the last column shows whether it is confirmed by the developers of the algorithms. ✔ indicates that developers have confirmed the logic flaws. We further analyze the mission failures and categorize them into four different types as follows (in the gray shaded rows):

1. *Crash between victim drones:* A victim drone is crashed into another victim drone.
2. *Crash into external objects:* A victim drone is crashed into an external object (not a victim drone).
3. *Suspended progress:* A swarm could not make meaningful progress, failing to complete the mission.
4. *Slow progress:* A swarm's progress is exceptionally slow, eventually failing to complete the mission in time. This is less severe than the suspended progress since the swarm may finish the mission if given a longer time.

Root Causes and Potential Fixes. We identify root causes of the mission failures and potential fixes via manual analysis. Note that all the fixes we present below resolved the problem in the tested scenarios. We also communicate with the developers to confirm the fixes. Fixes with '*(Confirmed)*' are the ones that are confirmed. We present a few cases in this section, and the remainings are in § IX-F.

C1-1. *Missing collision detection:* In A1, a leader drone does not have logic for avoiding other drones in a swarm. The algorithm developers confirmed that they thought that leader drones always move ahead of other drones, believing the logic is unnecessary. Details are in § V-E1.

> *Fix (Confirmed):* We reuse code snippets from a follower drone that detects other victim drones for the leader drone.

C1-2. *Naive multi-force handling:* A1 uses the artificial potential field (APF) to implement the drones' collision avoidance mechanism. Unfortunately, it has difficulty handling multiple forces are involved, as shown in § III's Fig. 3.

> *Fix (Confirmed):* We find that this is a fundamental weakness of the APF. One may reconfigure the algorithm to make the drone sense external objects earlier by changing the value of `influence_radius` (from 0.15 to 0.3). This will avoid a drone surrounded by external objects.

[2] *CX-Y* means "the root cause *Y* of a logic flaw in algorithm *X* (AX)"

C1-4. *Excessive force in APF:* A1 uses the artificial potential field (APF) to make drones' decisions at runtime. If a drone is at a location that is very far from the other drones in a swarm, a force to move toward the swarm becomes excessively strong, making the detached drone fly directly to the swarm without considering external objects on the path (e.g., wall). In other words, the drone decides to fly toward the wall because the force for rejoining the swarm becomes bigger than the force preventing the drone from crashing into the wall.

> *Fix (Confirmed):* We define a maximum value for all forces and assign a much larger value than the maximum value for the force related to obstacles (e.g., the wall). It requires changing 6 SLOC. This prevents the drone from crashing into obstacles but often causing the swarm stuck as described in C1-5, requiring the fix from C1-5 as well.

C1-5. *Naive swarm's pose measurement:* A1 measures the current pose of the entire swarm by computing the centroid of all drones. Unfortunately, this often neglects drones to fall behind significantly, eventually making the swarm unable to progress. Details are shown in § V-E2.

> *Fix (Confirmed):* We add code snippets (2 SLOC) to consider the drone's distances from the centroid, and if a drone is significantly far behind than others (e.g., more than two times), we make the leader wait for the other drones.

C2-1. *Overly-sensitive object detection:* Drones are configured to be overly sensitive in avoiding external objects, leading to crashes to other victim drones to avoid objects.

> *Fix (Confirmed):* We relax the object detection by changing `DEFAULT_WEIGHT_COSTS` to 0.219 (from 0.319) in A2.

C2-2. *Indefinite wait for crashed drones:* A2 uses a bidding algorithm to distribute tasks to individual drones. The algorithm has a bug that it does not exclude crashed drones (hence unusable) from the bidding process. After assigning a task to an inactive crashed drone, the algorithm waits for the task completion indefinitely, suspending progress.

> *Fix (Confirmed):* We change the bidding algorithm (10 SLOC) to reclaim tasks from crashed drones.

C2-3. *Long deadline for an assigned task:* A2's bidding algorithm has an internal deadline for each task assigned to a drone. However, the deadline is too long. When an attack drone successfully prevents victim drones from completing tasks, the algorithm keeps waiting for the task.

> *Fix (Confirmed):* We change the deadline (`SEARCH_TIMEOUT_TIME`) shorter in A2. This effectively mitigates the delays caused by the adversarial drones in our scenario.

C2-4. *Drones detaching from a swarm:* We observe that malfunctioning drones are moving outside of the map, detaching themselves from the swarm. This is because drones do not have any tasks to bid (i.e., finished all the tasks) have no incentive to stay in the swarm. This significantly delays the swarm's progress since the algorithm still waits for the task completion by the malfunctioning drone.

> *Fix (Confirmed):* We increase the individual drone's incentive value for being a part of the swarm.

C3-1 and C4-1. *Naive detouring method:* In A3, when a drone encounters an obstacle, it tries to detour the obstacle by randomly selecting the alternative direction (i.e., angle) to fly. Unfortunately, if objects are approaching the drone from the randomly decided direction, the drone crashes. Moreover, this method also performs poorly for drones escaping from a complex structure, delaying the progress significantly.

> *Fix :* For A3, we add more randomness in choosing a direction for detouring by changing 8 SLOC. For A4, we find that the randomness in the detouring process overly affects the decision. Hence, we remove the random values involved in the process by changing 2 SLOC.

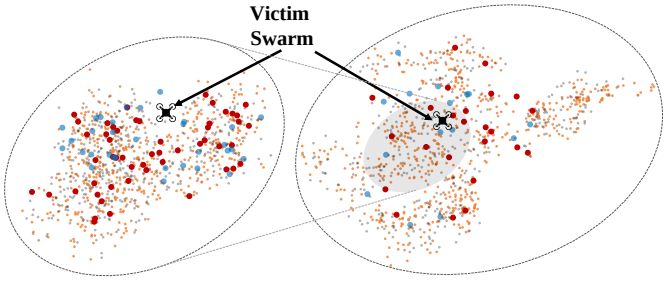
C4-2. *Detouring without sensing:* In A4, when a drone avoids an obstacle, it selects an alternative path. Unfortunately, it does not consider whether there is an obstacle in the alternative path. If there is an object in the path, the drone crashes. We present a detailed case study in § V-E3.

> *Fix :* We add 10 SLOC to make a drone sense the surroundings when it calculates an alternative path.

Quality of Fixes. To understand the quality of our fixes, we have applied them to the algorithms, and run SwarmFlawFinder on the fixed algorithms (for 24 hours per algorithm). The results show that the logic flaw targeted by the fix is no longer observed after applying each fix. Hence, we consider each fix successfully resolves its targeted logic flaw. Further, we apply *all the fixes together* (i.e., an integrated fix) and run SwarmFlawFinder to understand whether the integrated fix can eliminate all the logic flaws. We find that for A1, the integrated fix fails to resolve C1-2 and C1-6, because the fixes for C1-2 and C1-6 are conflicting. To solve this, we manually tune the configuration values (i.e., changing `influence_radius` to 0.225 and `repulsive_coef` to 300 in the fixes; the original fixes; the original fixes are changing them to 0.3 and 400), and the tuned integrated fix resolved all the logic flaws. Details can be found in § IX-D.

Side Effects of Fixes. While the fixes make the algorithms more robust, they may also cause overhead. We observe 3.9%, 2.5%, 1.2%, and 1.5% average overhead for A1, A2, A3, and A4, respectively. For the integrated fixes, we observe 11.4%, 9.0%, 2.2%, and 4.7% overhead for A1, A2, A3, and A4, respectively. Details can be found in [8]. Note that we do not observe fixes introducing additional logic flaws.

Impact of Flaws. In A1, C1-1~C1-4 are the most critical bugs since they will result in crashed drones. C1-5~C1-6 lead to mission delays, and the victim drones are intact; hence their impact is limited. In A2, A3, and A4, the crashes between drones are less critical than crashes in A1 since there are many victim drones, and crashing a few drones may not immediately lead to mission failures. However, since a crash in A2 (C2-2) can suspend the search progress, it is more critical than the crashes in A3 and A4. Slow progress type bugs in all algorithms are less impactful than other types of bugs.



(a) Visualized test cases generated for **A1** by SwarmFlawFinder

(b) Visualized test cases generated for **A1** by the random testing approach

Fig. 8. Spatial Distribution of Test cases generated by (a) SwarmFlawFinder and (b) the random testing approach.

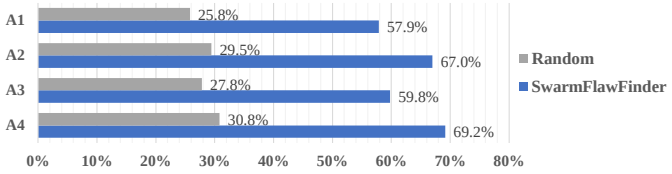


Fig. 9. Coverage of Unique Dcc Values.

C. Effectiveness of Dcc in Fuzz Testing

1) Creating Random Testing Approach: To understand the effectiveness of Dcc based guidance during the fuzz testing, we create a random testing approach by removing Dcc based guidance from SwarmFlawFinder. The random testing version only leverages the result of the execution (whether the mission is failed or not). If a test run resulted in a mission failure, it prioritizes similar tests by perturbing the test case with a small delta. If a test did not lead to a mission failure, it tries to mutate the test case with a larger random value, as SwarmFlawFinder does when it observes a similar Dcc value described in § IV-C.

2) Spatial Distribution of Test-cases: We run the random testing approach and SwarmFlawFinder on our evaluated algorithms for 24 hours to measure the spatial distribution of the test cases generated by the two techniques. Fig. 8 shows the results of A1 (Results for A2, A3, and A4 are presented in [8]). Specifically, Fig. 8-(a) is the results from SwarmFlawFinder while (b) is from the random testing approach. The silver round dotted circles approximately show the size of the area explored during the testing. Each dot in the figure represents a test case. Large dots indicate they result in new unique Dcc values, where small dots are not. Red and orange dots are the test cases that caused mission failures (i.e., discovering logic flaws). Silver and blue dots are the test cases that do not cause mission failures. Note that we do not limit searching space for both SwarmFlawFinder and random approach, and the results show that SwarmFlawFinder does more focused searching. The shaded area in Fig. 8-(b) represents the explored area by SwarmFlawFinder in (a).

Observations. First, SwarmFlawFinder is able to focus on testing a smaller but more promising area, as shown in the shaded area. Moreover, while it tests a smaller area, SwarmFlawFinder's test cases result in more new unique Dcc values (represented by the large red and blue dots). This is because, in part, SwarmFlawFinder can run more test cases exhaustively in the focused area, guided by Dcc, without any domain knowledge in finding the area. The random testing approach does not have such a particular focused area observed. Second, SwarmFlawFinder found on average 25.75% more failures than random testing (red and orange dots in Fig. 8), when we run both for the same period (i.e., 24 hours). We present details of the statistics in the Appendix (Fig. 16). Third, the random testing seems to find some unique Dcc values from the places that SwarmFlawFinder did not test (the large red and blue dots outside the shade). However, we manually check them and find that they are variants of the tests generated by SwarmFlawFinder, meaning that they are all subsets of SwarmFlawFinder's tests.

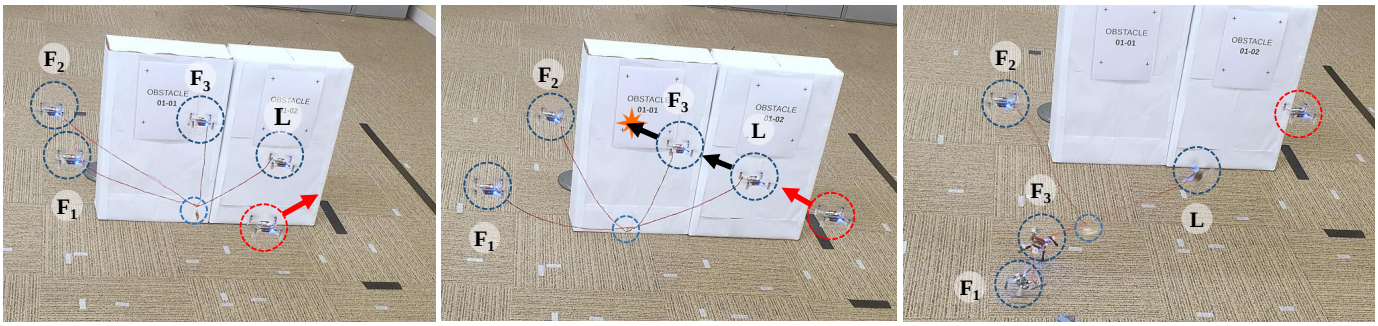
3) Impact of Searching Space on Random Testing: Observe that the random testing approach's test cases are spread over the wide area in Fig. 8. This is because the random testing approach lacks the guidance metric which is Dcc in SwarmFlawFinder. To further understand the effectiveness of Dcc and the impact of searching space, we conduct additional experiments with different searching spaces restrictions on random testing approach. Specifically, we run the random testing with the explored space (e.g., the gray shaded area in Fig. 8-(b)) obtained by SwarmFlawFinder. We also run two more experiments with 2x and 3x Base searching spaces (as shown in Fig. 17). The results show that the random testing performs better when given the searching space. However, it still misses three logic flaws C1-2, C1-3, and C2-1, that are dependent on the subtle timing. Details including all the experiment results (in Table IV) can be found in § IX-C2.

D. Coverage based on Dcc

We measure the coverage of Dcc values by SwarmFlawFinder. Specifically, we first collect an almost complete range of the Dcc values by running tests with attack drones on every 0.2 meters in the 3D space. Then, we run SwarmFlawFinder for 24 hours to understand how many Dcc values (out of the collected values) are covered. We also run the random testing version of SwarmFlawFinder (without the Dcc guidance) and measure the coverage of Dcc values. As shown in Fig. 9, SwarmFlawFinder covers two times more Dcc values (avg. 63.5%) than the random testing version (avg. 28.5%). Details can be found in § IX-B.

E. Case Studies

1) Missing Collision Detection in Adaptive Swarm: Fig. 10 shows three screenshots of a failed mission which we reproduced in the lab with real world drones. The failed mission represents the 'C1-1' in Table III. In Fig. 10-(a), the attack drone (red circled) approaches the leader drone (L), making it to move closer to another drone near the wall (F_3). In (b), the attack drone pushes the leader drone further. Interestingly, we find that the leader does not consider the fact that there is F_3, pushing it to the wall until F_3 crashes. In (c), after the crash, the attack drone still is alive.

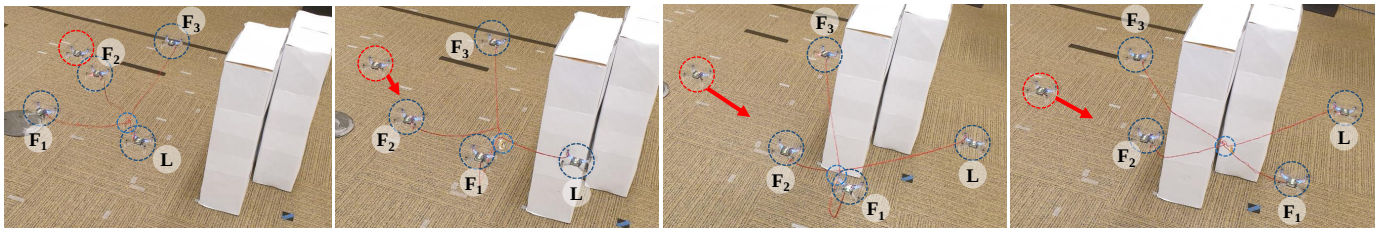


(a) Attack drone pushes the leader drone

(b) Leader drone moves back without considering the F_3, making F_3 crashing into the wall

(c) Mission failed with a crash (The attack drone still alive)

Fig. 10. Attack drone causing a victim drone (F_3) to crash into the wall.



(a) Attack drone chases a victim drone

(b) The chased victim drone blocks the other drone's way, making it stuck behind the wall

(c) Other drones make progress

(d) The entire swarm cannot make progress due to the drone stuck behind the wall

Fig. 11. Attack drone pushes a victim drone F_2 to suspend the swarm's progress.

(a) Victim drones approaching the corner

(b) Victim drones try to detour without considering the surrounding

Fig. 12. Drones crashing while detouring due to obstacles.

Analysis. We inspect the DCC values of the *leader drone before the crash*. Interestingly, its DCC values do not include other victim drones, even if they are very close. This means that the leader drone does not recognize and try to avoid other victim drones. We inspect the source code of A1 and find that it does not have the logic to detect other victim drones as external objects. The algorithm's developer confirms that the logic is omitted, because the leader drone will mostly fly ahead of other drones, making the mission failure difficult to be revealed without SWARMFLAWFINDER. We ran SWARMFLAWFINDER without the DCC guided feedback (i.e., random testing approach) for 24 hours and did not find the error.

2) Suspended Swarm Mission due to a Logic Flaw: We find another logic flaw (C1-5 in Table III) in A1. Fig. 11 shows the mission failure reproduced with the real-world drones. In (a), the attack drone (red circled) chases the victim drone F_2, making it go faster. This results in F_2 blocking the path of F_3. As shown in (b), F_3 is stalled because F_2 is going faster than expected. In (c), F_3 is completely behind the wall, while L and F_1 make progress toward the destination. Finally, in (d), due to the F_3, the other drones cannot make progress while F_3 cannot proceed due to the F_2 blocking its path.

Analysis. We manually analyze the algorithm to understand why the leader drone keeps moving forward while F_3 stays behind the wall. It turns out that the algorithm computes the centroid of all drones to measure the current position of the swarm. As long as the centroid is not falling behind, the leader keeps moving forward. Hence, even if F_3 cannot progress, the other drones' progress makes the centroid move toward the destination, giving the leader a wrong perception that the swarm is progressing. A possible fix is to consider the distance between the centroid and individual drones.

3) Detouring without Sensing: Fig. 12 shows the failed mission (C4-2 in A4): (a) the attack drone (red drone) pushes two victim drones into the corner. (b) The victim drones sense the corner and try to fly in the opposite direction. Then, both drones fly to the same location, causing a crash.

Analysis. This crash happens when an attack drone pushes multiple drones into the corner, making both of them try to escape from the corner. From our manual analysis, we find that the algorithm does not have code for detecting obstacles when detouring. As a result, when it computes a flight path to detour, it does not consider any obstacles in the path. We believe this is a mistake, and we resolved this issue by implementing sensing during detouring by reusing the existing code.

VI. DISCUSSION

Additional Attack Strategies. We acknowledge that there can be more sophisticated attack strategies, which may improve the SWARMFLAWFINDER's performance. Adding new attack strategies is straightforward. One can define a new attack behavior relative to a victim drone. The essence of this research is to show the feasibility of DCC based fuzz testing.

DCC and Behavior Abstraction. While it turns out DCC is highly effective in guiding the SWARMFLAWFINDER's testing

process, we do not argue that DCC is a direct abstraction of the swarm behavior. Instead, it is an approximation of the abstraction. However, we argue that it captures the behavior differences of swarm algorithms effectively.

VII. Related Work

Testing for Robotics. While systematic testing for robotics systems helps improve the overall quality and safety of the systems significantly, testing robots in real-world conditions is often expensive and unsafe. As a result, simulation-based approaches have been widely adopted in robotics testing [39]–[46], and shown to be effective [47]. [44] proposes coverage-driven verification (CDV) for evaluating the testing progress of the system under test. CDV and DCC in SWARMFLAWFINDER share the same goal while CDV is coarse-grained and requires definitions from developers. [48], [49] apply combinatorial interaction testing to detect flaws triggered by interactions of parameters, while they also require definitions of systems' configuration space. Calò [50] proposes using search-based approach to generate collision inducing configurations for autonomous driving systems. [42] integrates dynamic physical models of the robot to generate physically valid yet stressful test cases. SWARMFLAWFINDER targets swarm robotics, which is more complex than individual robots. [51] aims to find faults in a flocking algorithm of on ground vehicle swarms by using genetic algorithms (GA) [52]. However, they are not applicable to the non-flocking swarm algorithms, which require more sophisticated definitions such as fitness functions. Specifically, their fitness function focuses on handling flocking algorithms, considering splitting swarms as failures. However, A3 in our paper dynamically forms and splits swarms to improve the efficiency of searching. Hence, a perfectly fine mission of A3 can be considered a failure. The idea of GA can be applied to SWARMFLAWFINDER.

Formal validation and verification for robotics systems have been studied [53]–[58]. However, they require fine-grained definitions of correct behaviors, which typically need to be defined by domain experts. SWARMFLAWFINDER only requires a high-level failure definition (e.g., 200% of typical deadline).

Fuzz Testing. Fuzz testing has become widely used today due to its effectiveness. Some of these studies aim to improve the coverage-driven [9]–[11] fuzzers, while others [1], [59]–[62] aim to retrieve more advanced information (e.g., code- and data-flow) to handle systems on new domains/platforms or improve input mutation strategy. Hybrid fuzzing techniques [1], [63], [64] are proposed to increase testing coverage using both dynamic and symbolic execution. Conventional techniques that rely on obvious symptoms of program failures (e.g., segmentation faults) in detecting bugs and exercising new unique execution paths are ineffective to swarm robotics because traditional coverage metrics are not effective for swarm robotics. SWARMFLAWFINDER proposes and leverages the degree of the causal contribution (instead of code coverage) to effectively guide the testing process.

Fuzz Testing for Drones. There are several fuzzers targeting drones [65]–[70]. However, they are designed to find vulnerabilities in a single drone (not from swarm robotics). Note that they (i.e., fuzzers for a single drone) can replace the adversarial drone in our approach, and it is complementary to our paper. Moreover, existing fuzzers [65]–[70] try to find bugs in a target device's software (e.g., firmware), assuming a stronger attack model than ours. Our threat model assumes no direct access to the drones. Lastly, existing fuzzers have limited scope in the types of bugs they are targeting. [68]–[70] aim to detect general type bugs only (e.g., buffer overflow). [67] can only detect limited types of misbehavior (e.g., finding input validation bugs). [65] relies on substantial domain knowledge, which is not designed for swarm robotics. Others [66], [69], [70] also focus on bugs related to a specific environment, such as weak ports [66], MAVLink protocol [69], and WiFi [70]. However, our approach can be used to detect a wide range of bugs in various swarm algorithms unlike those existing specific environments, general type, and implementation-oriented bugs. Moreover, SWARMFLAWFINDER can detect logic flaws without requiring particular domain expertise in drone swarm fuzz testing, as we use DCC to abstract swarm behaviors.

Attacks and Defences for Drones. As drones are getting more attention in the research and industry communities, attacks [71]–[73] and defenses [74]–[80] of drones have gained significant attention. There are testing tools [81] developed to run various known attacks (e.g., GPS spoofing, jamming, and acoustic attacks) against drones. Compared to the previous work which focuses on individual drones, SWARMFLAWFINDER focuses on finding logic flaws in drone swarm algorithms. To the best of our knowledge, this is the first work that finds logic flaws of the swarm robotics algorithms.

VIII. Conclusion

This paper develops a novel fuzz testing approach for swarm robotics, SWARMFLAWFINDER, to discover swarm algorithms' logic flaws. We propose a novel concept of *the degree of the causal contribution* and use it as a feedback metric for fuzz testing. Our extensive evaluation with four swarm algorithms shows that SWARMFLAWFINDER is highly effective, finding 42 unique previously unknown logic flaws (all of them have been confirmed by the developers). We release the code and data for future research.

Acknowledgment

We thank the anonymous referees for their constructive feedback. The authors gratefully acknowledge the support of NSF #1916499, #1908021, and #1850392. This work was also supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant by the Korea government (MSIT) (No. 2021-0-01817, Development of Next-Generation Computing Techniques for Hyper-Composable Datacenters) and Basic Science Research Program by the National Research Foundation of Korea (2021 R1F1A1049822). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

REFERENCES

[1] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019.

[2] Y. Wang, P. Jia, L. Liu, C. Huang, and Z. Liu, "A systematic review of fuzzing based on machine learning techniques," *PloS one*, 2020.

[3] P. Wang, X. Zhou, K. Lu, T. Yue, and Y. Liu, "The progress, challenges, and perspectives of directed greybox fuzzing," *arXiv preprint arXiv:2005.11907*, 2020.

[4] R. Agishev, "Adaptive Control of Swarm of Drones for Obstacle Avoidance," Master's thesis, Skolkovo Institute of Science and Technology, 2019.

[5] P. Henderson, M. Vertescher, D. Meger, and M. Coates, "Cost adaptation for robust decentralized swarm behaviour," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018.

[6] M. G. Cimino, M. Lega, M. Monaco, and G. Vaglini, "Adaptive exploration of a uavs swarm for distributed targets detection and tracking." in *ICPRAM*, 2019.

[7] P. Carnelli, "SwarmRoboticsSim," 2017, https://github.com/pc0179/SwarmRoboticsSim.

[8] SwarmFlawFinder, "Project Website," 2021, https://github.com/adswarm/src.

[9] Google, "syzkaller is an unsupervised, coverage-guided kernel fuzzer," https://github.com/google/syzkaller, 2018.

[10] LLVM, "LibFuzzer: a library for coverage-guided fuzz testing," https://llvm.org/docs/LibFuzzer.html, 2021.

[11] M. Zalewski, "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl.

[12] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *Network and Distributed System Security Symposium (NDSS)*, 2008.

[13] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, 2018.

[14] B. Beizer, *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995.

[15] A. F. Winfield, C. J. Harper, and J. Nembrini, "Towards dependable swarms and a new discipline of swarm engineering," in *International Workshop on Swarm Robotics*. Springer, 2004.

[16] I. Sargeant and A. Tomlinson, "Modelling malicious entities in a robotic swarm," in *2013 IEEE/AIAA 32nd Digital Avionics Systems Conference (DASC)*. IEEE, 2013.

[17] F. Higgins, A. Tomlinson, and K. M. Martin, "Threats to the swarm: Security considerations for swarm robotics," *International Journal on Advances in Security*, 2009.

[18] C. Taylor, A. Siebold, and C. Nowzari, "On the effects of minimally invasive collision avoidance on an emergent behavior," in *International Conference on Swarm Intelligence*. Springer, 2020.

[19] H. Hamann and H. Wörn, "A framework of space–time continuous models for algorithm design in swarm robotics," *Swarm Intelligence*, 2008.

[20] C. Harper and A. Winfield, "Direct lyapunov design - a synthesis procedure for motor schema using a second-order lyapunov stability theorem," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2002.

[21] A. J. Kerns, D. P. Shepard, J. A. Bhatti, and T. E. Humphreys, "Unmanned aircraft capture and control via gps spoofing," *Journal of Field Robotics*, 2014.

[22] S.-H. Seo, B.-H. Lee, S.-H. Im, and G.-I. Jee, "Effect of spoofing on unmanned aerial vehicle using counterfeited gps signal," *Journal of Positioning, Navigation, and Timing*, 2015.

[23] D. Lewis, *Counterfactuals*. Oxford: Blackwell Publishers, 1973.

[24] Y. Kwon, D. Kim, W. N. Sumner, K. Kim, B. Saltaformaggio, X. Zhang, and D. Xu, "LDX: Causality inference by lightweight dual execution," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*.

[25] J. P. Lewis, "Fast normalized cross-correlation," in *Proceedings of the Vision Interface*, 1995.

[26] L. Yu and V. Giurgiutiu, "Advanced signal processing for enhanced damage detection with embedded ultrasonics structural radar using piezoelectric wafer active sensors," in *Smart Structures & Systems – An International Journal of Mechatronics, Sensors, Monitoring, Control, Diagnosis, and Maintenance*, 2005.

[27] D. M. Tsai and C. T. Lin, "The evaluation of normalized cross correlations for defect detection," *Pattern Recognition Letters*, 2003.

[28] E. Rafajłowicz, M. Wnuk, and W. Rafajłowicz, "Local detection of defects from image sequences." *International Journal of Applied Mathematics & Computer Science*, 2008.

[29] C. Howard, "Algorithms developed to make drone swarm move together," 2020, https://github.com/choward1491/SwarmAlgorithms.

[30] T. Vicsek, "Autonomous mission control of drone flocks," EOTVOS Lorand Tudomanyegetem Budapest Hungary, Tech. Rep., 2019.

[31] J. S. Huang, S. Ma, G. Li, O. W. Yang, and C. Shao, "An artificial swan formation using the finsler measure in the dynamic window control," *Int J Swarm Evol Comput*, 2020.

[32] B. Balázs, G. Vásárhelyi, and T. Vicsek, "Adaptive leadership overcomes persistence–responsivity trade-off in flocking," *Journal of the Royal Society Interface*, 2020.

[33] L. Ma, W. Bao, X. Zhu, M. Wu, Y. Wang, Y. Ling, and W. Zhou, "O-flocking: Optimized flocking model on autonomous navigation for robotic swarm," in *International Conference on Swarm Intelligence*, 2020.

[34] A. Wright, "swarmSimRescue," 2014, https://github.com/aywrite/swarmSimRescue.

[35] J. Harwell and M. Gini, "Improved swarm engineering: Aligning intuition and analysis," *arXiv preprint arXiv:2012.04144*, 2020.

[36] K. Patel, "optimization-wolf-search-algorithm," 2017, https://github.com/bavalia/optimization-wolf-search-algorithm.

[37] R. Berg, "Zebro-Search-and-Rescue," 2020, https://github.com/RobvandenBerg/Zebro-Search-and-Rescue.

[38] G. M. Fricke, J. P. Hecker, A. D. Griego, L. T. Tran, and M. E. Moses, "A distributed deterministic spiral search algorithm for swarms," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016.

[39] C. Aguero, N. Koenig, I. Chen, H. Boyer, S. Peters, J. Hsu, B. Gerkey, S. Paepcke, J. Rivero, J. Manzo, E. Krotkov, and G. Pratt, "Inside the virtual robotics challenge: Simulating real-time robotic disaster response," *Automation Science and Engineering, IEEE Transactions on*, 2015.

[40] A. Patelli and L. Mottola, "Model-based real-time testing of drone autopilots," in *Proceedings of the 2nd Workshop on Micro Aerial Vehicle Networks, Systems, and Applications for Civilian Use*, 2016.

[41] M. Lindvall, A. Porter, G. Magnusson, and C. Schulze, "Metamorphic model-based testing of autonomous systems," in *Proceedings of the 2nd International Workshop on Metamorphic Testing*, 2017.

[42] C. Hildebrandt, S. Elbaum, N. Bezzo, and M. B. Dwyer, "Feasible and stressful trajectory generation for mobile robots," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2020.

[43] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and service robotics*, 2018.

[44] D. Araiza-Illan, D. Western, A. G. Pipe, and K. Eder, "Systematic and realistic testing in simulation of control code for robots in collaborative human-robot interactions," in *Annual Conference Towards Autonomous Robotic Systems*, 2016.

[45] C. Hutchison, M. Zizyte, P. E. Lanigan, D. Guttendorf, M. Wagner, C. Le Goues, and P. Koopman, "Robustness testing of autonomy software," in *IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2018.

[46] A. Afzal, C. Le Goues, M. Hilton, and C. S. Timperley, "A study on challenges of testing robotic systems," in *IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020.

[47] C. S. Timperley, A. Afzal, D. S. Katz, J. M. Hernandez, and C. Le Goues, "Crashing simulated planes is cheap: Can simulation detect robotics bugs early?" in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 331–342.

[48] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, 2011.

[49] D. R. Kuhn, J. M. Higdon, J. F. Lawrence, R. N. Kacker, and Y. Lei, "Combinatorial methods for event sequence testing," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012.

[50] A. Calò, P. Arcaini, S. Ali, F. Hauer, and F. Ishikawa, "Simultaneously searching and solving multiple avoidable collisions for testing

autonomous driving systems," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, 2020.
[51] H. Wei, J. Timmis, and R. Alexander, "Evolving test environments to identify faults in swarm robotics algorithms," in *IEEE Congress on Evolutionary Computation (CEC)*, 2017.
[52] M. Dorigo, V. Trianni, E. Şahin, R. Groß, T. H. Labella, G. Baldassarre, S. Nolfi, J.-L. Deneubourg, F. Mondada, D. Floreano *et al.*, "Evolving self-organizing behaviors for a swarm-bot," *Autonomous Robots*, 2004.
[53] S. Bensalem, L. de Silva, F. Ingrand, and R. Yan, "A verifiable and correct-by-construction controller for robot functional levels," *arXiv preprint arXiv:1309.0442*, 2013.
[54] A. Desai, S. Qadeer, and S. A. Seshia, "Programming safe robotics systems: Challenges and advances," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2018.
[55] R. C. Cardoso, L. A. Dennis, M. Farrell, M. Fisher, and M. Luckcuck, "Towards compositional verification for modular robotic systems," *Electronic Proceedings in Theoretical Computer Science*, 2020.
[56] X. Zheng, C. Julien, M. Kim, and S. Khurshid, "On the state of the art in verification and validation in cyber physical systems," *The University of Texas at Austin, The Center for Advanced Research in Software Engineering, Tech. Rep. TR-ARiSE-2014-001*, 2014.
[57] R. C. Cardoso, M. Farrell, M. Luckcuck, A. Ferrando, and M. Fisher, "Heterogeneous verification of an autonomous curiosity rover," in *NASA Formal Methods*. Springer International Publishing, 2020.
[58] H. T. Dinh and T. Holvoet, "A framework for verifying autonomous robotic agents against environment assumptions," in *Advances in Practical Applications of Agents, Multi-Agent Systems, and Trustworthiness. The PAAMS Collection*. Springer International Publishing, 2020.
[59] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "Parmesan: Sanitizer-guided greybox fuzzing," in *29th USENIX Security Symposium*, 2020.
[60] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.
[61] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. de Ruiter, K. Sagonas, and J. Somorovsky, "Analysis of DTLS implementations using protocol state fuzzing," in *29th USENIX Security Symposium*, 2020.
[62] H. Kim, J. Lee, E. Lee, and Y. Kim, "Touching the untouchables: Dynamic security analysis of the lte control plane," in *IEEE Symposium on Security and Privacy (SP)*, 2019.
[63] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium*, 2018.
[64] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "Savior: Towards bug-driven hybrid testing," in *IEEE Symposium on Security and Privacy (SP)*, 2020.
[65] H. Kim, M. O. Ozmen, A. Bianchi, Z. B. Celik, and D. Xu, "Pgfuzz: Policy-guided fuzzing for robotic vehicles."
[66] D. Rudo, D. Zeng *et al.*, "Consumer UAV Cybersecurity Vulnerability Assessment Using Fuzzing Tests," *arXiv:2008.03621*, 2020.
[67] 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*, 2019.
[68] O. M. Alhawi, M. A. Mustafa, and L. C. Cordeiro, "Finding security vulnerabilities in unmanned aerial vehicles using software verification," *arXiv preprint arXiv:1906.11488*, 2019.
[69] K. Domin, I. Symeonidis, and E. Marin, "Security analysis of the drone communication protocol: Fuzzing the mavlink protocol," 2016.
[70] M. Hooper, Y. Tian, R. Zhou, B. Cao, A. P. Lauf, L. Watkins, W. H. Robinson, and W. Alexis, "Securing commercial wifi-based uavs from common security attacks," in *MILCOM 2016-2016 IEEE Military Communications Conference*, 2016.
[71] J. Valente and A. A. Cardenas, "Understanding security threats in consumer drones through the lens of the discovery quadcopter family," in *Proceedings of the 2017 Workshop on Internet of Things Security and Privacy*, 2017.
[72] Y. Son, H. Shin, D. Kim, Y. Park, J. Noh, K. Choi, J. Choi, and Y. Kim, "Rocking drones with intentional sound noise on gyroscopic sensors," in *24th USENIX Security Symposium*, 2015.
[73] I. Pustogarov, T. Ristenpart, and V. Shmatikov, "Using program analysis to synthesize sensor spoofing attacks," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. Association for Computing Machinery, 2017.
[74] H. Choi, W.-C. Lee, Y. Aafer, F. Fei, Z. Tu, X. Zhang, D. Xu, and X. Deng, "Detecting attacks against robotic vehicles: A control invariant approach," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.
[75] R. Quinonez, J. Giraldo, L. Salazar, E. Bauman, A. Cardenas, and Z. Lin, "SAVIOR: Securing autonomous vehicles with robust physical invariants," in *29th USENIX Security Symposium*, 2020.
[76] N. Moustafa and A. Jolfaei, "Autonomous detection of malicious events using machine learning models in drone networks," in *Proceedings of the 2nd ACM MobiCom Workshop on Drone Assisted Wireless Communications for 5G and Beyond*, 2020.
[77] R. Mitchell and I.-R. Chen, "Adaptive intrusion detection of malicious unmanned air vehicles using behavior rule specifications," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*.
[78] R. R. Beck, A. Vijeev, and V. Ganapathy, "Privaros: A framework for privacy-compliant delivery drones," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[79] S.-J. Chung, A. A. Paranjape, P. Dames, S. Shen, and V. Kumar, "A survey on aerial swarm robotics," *IEEE Transactions on Robotics*, 2018.
[80] A. A. Paranjape, S.-J. Chung, K. Kim, and D. H. Shim, "Robotic herding of a flock of birds using an unmanned aerial vehicle," *IEEE Transactions on Robotics*, 2018.
[81] M. S. bin Mohammad Fadilah, V. Balachandran, P. Loh, and M. Chua, "DRAT: A drone attack tool for vulnerability assessment," in *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, 2020.
[82] DARPA, "OFFensive Swarm-Enabled Tactics (OFFSET)," 2017, https://www.darpa.mil/work-with-us/offensive-swarm-enabled-tactics.
[83] DARPAtv, "Teams Test Swarm Autonomy in Second Major OFFSET Field Experiment," 2019, https://www.youtube.com/watch?v=ruWC10AW87E.
[84] D. Hambling, "What Are Drone Swarms And Why Does Every Military Suddenly Want One?" 2021, https://www.forbes.com/sites/davidhambling/2021/03/01/what-are-drone-swarms-and-why-does-everyone-suddenly-want-one/?sh=2a5f085d2f5c.
[85] K. N. McGuire, C. De Wagter, K. Tuyls, H. J. Kappen, and G. C. H. E. de Croon, "Minimal navigation solution for a swarm of tiny flying robots to explore an unknown environment," 2019.
[86] T. Delft, "SGBA-code," 2020, https://github.com/tudelft/SGBA_code_SR_2019.

IX. Appendix

A. Algorithm Selection

1) Selection Criteria: As shown in Fig. 13-❶, we exhaustively search all publicly accessible swarm algorithms (i.e., 46 algorithms in the second row, ❷) and select the reproducible ones (i.e., 26 algorithms in the third row, ❸).

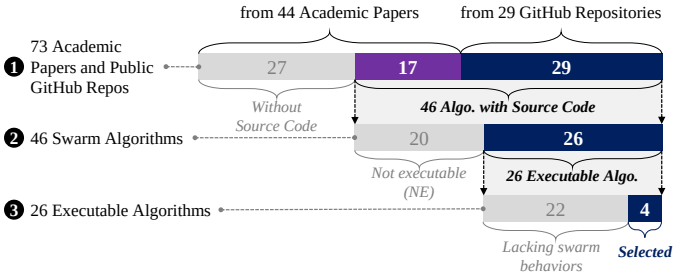


Fig. 13. Algorithm Selection Process

Not Executable Algorithms. During the process, we encounter 20 swarm algorithms that are not executable due to various reasons, including compilation errors (e.g., missing libraries/packages), runtime errors, and missing modules. Summary of errors for each algorithm can be found in [8].

Algorithms lacking Swarm Behaviors. We further inspect the 26 executable algorithms and prune out 22 algorithms lacking

swarm behaviors. Specifically, 21 algorithms do not exhibit communications between drones in the swarm, meaning that a drone will consider other drones as merely an external object to avoid. 16 algorithms do not allow us to introduce external attack drones; hence we prune out them. 2 algorithms are immature, meaning that they fail on provided example missions without any interventions. We focus on algorithms that at least can finish simple missions without errors. We further elaborate on the details of our analysis on [8].

Sizes of the Algorithms. Fig. 14 shows the SLOC of all the considered swarm algorithms' source code size in lines of code. We count the SLOC of swarm algorithms, excluding files for installations and configurations. It shows the selected algorithms' sizes are comparable to others and representative.

Commercial Swarm Algorithms. The reason that we do not have commercial swarm algorithms in our evaluation is that they are not publicly available for us to run. We comment that one of our selected swarm algorithms' authors mention that their recent version of the swarm algorithm is not publicly accessible due to legal issues. We could not investigate the details of those legal issues, but we believe that their codebase might be used in a proprietary product.

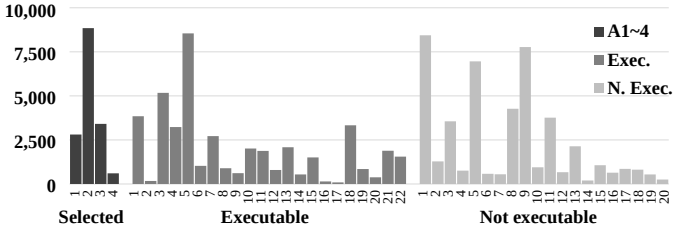


Fig. 14. SLOC of Considered and Selected Algorithms. Avg. of A1-4: 3,919 lines, Executable: 1,968 lines, and Not Executable: 2,305 lines.

2) Representativeness with respect to Real-world Examples: We believe our selection of the algorithms is comparable to the commercial algorithms because the four selected algorithms can conduct complex swarm scenarios that commercial swarms target. Specifically, we compare our selected algorithms with other publicly known swarm projects to understand the representativeness of our selection. In particular, DARPA's OFFSET program [82] conducted swarm missions aligned with our selected swarm algorithms: searching missions in urban/rural areas [83]. While the source code of their algorithms is not available, from the materials provided by DARPA, our algorithms A2 and A3 are comparable. Also, the column from Forbes [84] introduces the Reynolds' Boids model as the theoretical base for the modern military's swarm operation. A3 is comparable as it uses the same flocking model. Another popular swarm searching project by TU Delft [85] releases its source code [86]. We compare it with our algorithms, and it is smaller than A1, A2, and A3. Moreover, we believe that an up-to-date version of A4 [7] might be used in proprietary products, while the authors choose not to reveal the details.

B. Observed Unique Dcc Values

Fig. 15 shows the number of newly observed Dcc values over 12 hours of testing. Observe that most new Dcc values are discovered in the first 8-9 hours, showing the effectiveness of Dcc guided testing and justifying our 24 hours of timeout.

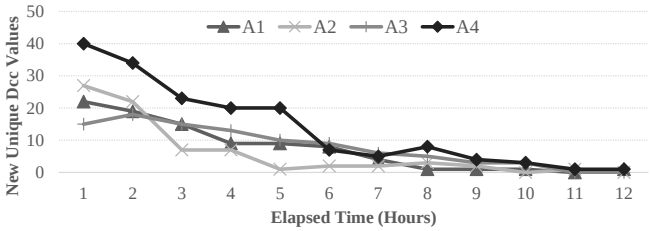


Fig. 15. Observed unique Dcc values during testing over time

C. Random Testing Approach vs SwarmFlawFinder

In § V-C, we created a random testing approach by removing the Dcc guidance from SwarmFlawFinder. We use the random testing approach to understand which components of SwarmFlawFinder make our approach more effective.

1) Effectiveness in Finding Mission Failures: Finding mission failures during testing is critical since they can lead to logic flaws of the algorithms. Fig. 16 shows the number of tests leading to mission failures executed by SwarmFlawFinder and a random testing approach (i.e., SwarmFlawFinder without the Dcc guidance). Observe that SwarmFlawFinder covers more test cases leading to mission failures. Note that the total number of tested missions is similar between the random testing and SwarmFlawFinder, because it depends on the execution time of each test case.

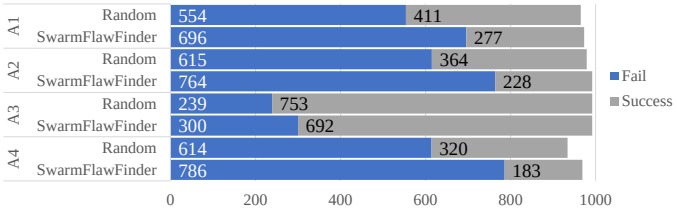
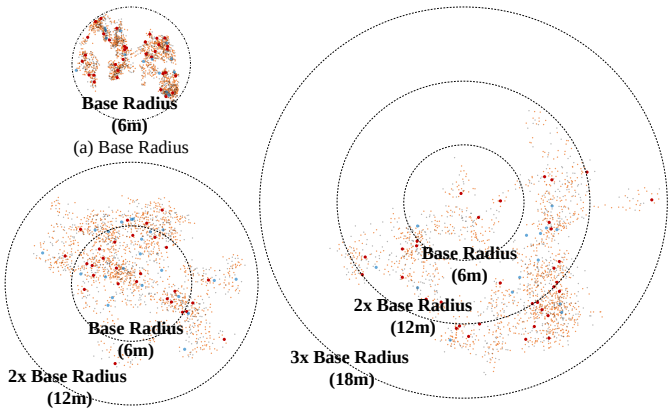


Fig. 16. Effective test cases (i.e., failures) from the random testing approach and SwarmFlawFinder

2) Impact of Searching Space on Random Testing Approach: SwarmFlawFinder's Dcc based guided fuzz testing prioritizes test cases generated in an area that can lead to more unique Dcc values (or exercise diverse swarm behaviors). In this experiment, we aim to understand the importance of finding the searching space in SwarmFlawFinder. Specifically, we run the random testing approach (which is essentially SwarmFlawFinder without Dcc guided testing) with different searching space restrictions, obtained by SwarmFlawFinder. Note that except for the searching spaces, we keep the original configurations described in § V-A2. We define three different searching spaces for each algorithm. First, we run SwarmFlawFinder and obtain the explored space by SwarmFlawFinder as shown in Fig. 17-(a), considering it the baseline space. Second, from the baseline space, we define 2x and 3x Base (Fig. 17-(b) and (c)) by extending the radius of the baseline by 2 and 3 times.



(a) Base Radius (b) 2x Base Radius (c) 3x Base Radius

Fig. 17. Examples of Searching Space Definition from A1. Dots in this figure represent executed test cases with the searching space restrictions.

Results. Table IV shows the experiments results. Observe the random testing approach's results vary depending on the searching space restrictions. First, without any searching space restriction ("No Restrict." column), the random approach misses many unique flaws (represented as red cells): missing 8 from A1, 4 from A2, 3 from A3, and 4 from A4. When we provide a restricted searching space (the space found by SWARMFLAWFINDER), the random approach finds 10 more flaws (4 for A1, 1 for A2, 1 for A3, and 4 for A4) than the random testing without the space restriction. If we simply look at the number of mission failures (not the unique failures), the random approach with the restriction finds even more instances than our system. However, the quality of testing is worse than ours. It misses 9 flaws (C1-1, C1-2, C2-1, C2-3, and C3-1).

Our manual analysis shows that those flaws are *dependent on subtle timings* (i.e., to expose the flaws, an attack drone has to approach from a certain pose when the swarm makes a turn). Without the guidance of DCC, the random testing approach has difficulty catch such subtle timings. This result shows that while the searching space is important in testing, DCC guided test mutation plays a critical role in finding subtle logical flaws. Note that *finding the searching space is a core contribution* of SWARMFLAWFINDER, which the random testing approach by itself *cannot* achieve.

Further, we run the experiments with 2x and 3x Bases, where they mostly perform worse as the searching space gets larger but still better than the one with no restriction. There are two exceptions in A1 (C1-2 and C1-4). With the 2x Base space, the random testing finds 1 more unique flaw in C1-2. Similarly, C1-4 is not found with the 2x Base space while found with the 3x Base space. Our manual analysis shows that the random testing approach's result is highly dependent on the randomness in test mutation.

D. Quality of Fixes

A fix is effective if SWARMFLAWFINDER fails to find a logical flaw the fix aims to resolve. In addition, we create an integrated fix that combines all the fixes in the algorithm to check whether fixes conflict with others. If there is no conflict, the integrated fix should eliminate all logical flaws we find.

TABLE IV
SWARMFLAWFINDER VS RANDOM TESTING, WITH RESPECT TO DIFFERENT SEARCHING SUBSPACE RESTRICTIONS.

ID	Root Cause	SWARMFLAWFINDER		Random Testing Approach							
		No Restrict.		No Restrict.		Base		2x Base		3x Base	
		# Exe.	Uq.	# Exe.	Uq.	# Exe.	Uq.	# Exe.	Uq.	# Exe.	Uq.
A1	**Crash btw. Drones**	**273**	**9**	**166**	**4**	**251**	**6**	**260**	**5**	**148**	**4**
	C1-1	86	4	49	3	80	3	85	3	28	3
	C1-2	176	4	117	1	162	2	175	2	120	1
	C1-3	11	1	0	0	9	1	0	0	0	0
	Crash into ext. objects	**435**	**8**	**359**	**5**	**407**	**7**	**375**	**5**	**348**	**5**
	C1-1	88	3	89	3	79	3	65	3	89	3
	C1-2	326	3	270	2	310	2	310	2	259	2
	C1-3	3	1	0	0	7	1	0	0	0	0
	C1-4	18	1	0	0	11	1	0	0	0	0
	Suspended progress	**671**	**2**	**594**	**2**	**752**	**2**	**708**	**2**	**617**	**2**
	C1-5	242	1	183	1	298	1	236	1	190	1
	C1-6	429	1	411	1	454	1	472	1	427	1
	Slow progress	**175**	**1**	**163**	**1**	**179**	**1**	**178**	**1**	**137**	**1**
	C1-6	175	1	163	1	179	1	178	1	137	1
	Total:	1,554	20	1,282	12	1,589	16	1,521	13	1,250	12
A2	**Crash btw. Drones**	**28**	**3**	**20**	**1**	**25**	**1**	**14**	**1**	**24**	**1**
	C2-1	28	3	20	1	25	1	14	1	24	1
	Suspended progress	**119**	**1**	**99**	**1**	**140**	**1**	**120**	**1**	**116**	**1**
	C2-2	119	1	99	1	140	1	120	1	116	1
	Slow progress	**608**	**4**	**415**	**2**	**592**	**3**	**524**	**2**	**421**	**2**
	C2-3	586	3	415	2	571	2	524	2	421	2
	C2-4	22	1	0	0	21	1	0	0	0	0
	Total:	755	8	534	4	757	5	658	4	561	4
A3	**Crash into ext. objects**	**47**	**2**	**50**	**1**	**36**	**1**	**38**	**1**	**46**	**1**
	C3-1	10	1	0	0	0	0	0	0	0	0
	C3-2	37	1	50	1	36	1	38	1	46	1
	Slow progress	**240**	**4**	**182**	**2**	**189**	**3**	**178**	**3**	**166**	**2**
	C3-1	23	2	16	1	36	1	28	1	22	1
	C3-2	217	2	166	1	153	2	150	2	144	1
	Total:	287	6	232	3	225	4	216	4	212	3
A4	**Crash btw. Drones**	**230**	**3**	**210**	**1**	**218**	**3**	**201**	**2**	**189**	**1**
	C4-1	216	1	210	1	207	1	193	1	187	1
	C4-2	14	2	0	0	11	2	7	1	0	0
	Crash into ext. objects	**630**	**3**	**411**	**1**	**461**	**3**	**431**	**2**	**411**	**1**
	C4-1	599	1	411	1	427	1	414	1	390	1
	C4-2	31	2	0	0	34	2	17	1	0	0
	Slow progress	**1,228**	**2**	**887**	**2**	**1,005**	**2**	**981**	**2**	**850**	**2**
	C4-3	1,228	2	887	2	1,005	2	981	2	850	2
	Total:	2,088	8	1,508	4	1,684	8	1,613	6	1,450	4

Invidivual Fixes for A1. Table V shows the results for A1. The numbers in the table represent the number of failed missions during the testing. The "Unpatched" columns show the SWARMFLAWFINDER's result on the original algorithm (identical to Table III). Observe that once each fix is applied, SWARMFLAWFINDER does not find any mission failures caused by the fixed logic flaw, meaning that individual fixes are effective. For instance, with the fix for C1-1, SWARMFLAWFINDER fails to find mission failures due to C1-1. A green cell represents a fix that successfully resolves the targeted flaw. Note that some fixes resolve flaws that are not targeted to handle. For example, the fix for C1-2 resolves flaws caused by C1-3 and C1-4 (represented as yellow cells). The fix for C1-6 resolves flaws of C1-3 and C1-4, because the fix for C1-2 makes drones more reactive, avoiding crashes due to C1-3 and C1-4. Similarly, the fix for C1-6 increases the sensing sensitivity, mitigating crashes caused by C1-3 and C1-4.

Integrated Fix for A1. The last column shows the result

TABLE V
FUZZ TESTING WITH FIXES FOR A1.

ID	Root Cause	Unpatched (Org.)		Fix for C1-1		Fix for C1-2		Fix for C1-3		Fix for C1-4		Fix for C1-5		Fix for C1-6		Integrated Fix	
		# Exec.	Uniq.	# Exec.	Uniq.	# Exec.	Uniq.	# Exec.	Uniq.	# Exec.	Uniq.	# Exec.	Uniq.	# Exec.	Uniq.	# Exec.	Uniq.
A1	**Crash btw. victim drones**	**273**	**9**	**152**	**5**	**26**	**4**	**261**	**8**	**271**	**9**	**279**	**9**	**36**	**8**	**0**	**0**
	C1-1	86	4	0	0	26	4	79	4	85	4	81	4	14	4	0	0
	C1-2	176	4	146	4	0	0	182	4	176	4	181	4	22	4	0	0
	C1-3	11	1	6	1	0	0	0	0	10	1	17	1	0	0	0	0
	Crash into ext. objects	**435**	**8**	**324**	**5**	**52**	**3**	**406**	**7**	**418**	**7**	**432**	**8**	**90**	**6**	**0**	**0**
	C1-1	88	3	0	0	52	3	77	3	81	3	79	3	44	3	0	0
	C1-2	326	3	315	3	0	0	309	3	331	3	331	3	46	3	0	0
	C1-3	3	1	5	1	0	0	0	0	6	1	7	1	0	0	0	0
	C1-4	18	1	4	1	0	0	20	1	0	0	15	1	0	0	0	0
	Suspended progress	**671**	**2**	**636**	**2**	**631**	**2**	**683**	**2**	**648**	**2**	**553**	**1**	**453**	**1**	**101**	**2**
	C1-5	242	1	224	1	317	1	243	1	229	1	0	0	453	1	79	1
	C1-6	429	1	412	1	314	1	440	1	419	1	553	1	0	0	22	1
	Slow progress	**175**	**1**	**181**	**1**	**112**	**1**	**175**	**1**	**168**	**1**	**240**	**1**	**0**	**0**	**3**	**1**
	C1-6	175	1	181	1	112	1	175	1	168	1	240	1	0	0	3	1
	Total:	1,554	20	1,293	13	821	10	1,525	18	1,505	19	1,504	19	579	15	104	3

Green: Fixes resolve targeted flaws, **Yellow**: Fixes resolve additional non-targeted flaws, **Red**: Fixes fail to resolve targted flaws.

from the integrated fix. It resolves the flaws from C1-1 to C1-4. However, it fails to handle C1-5 and C1-6. Our manual analysis points out that the fixes for C1-5 and C1-6 are conflicting. Specifically, the fix for C1-5 makes drones move together, waiting for slower drones if needed. However, the fix for C1-6 makes drones sensitive in avoiding obstacles. To this end, when there is an obstacle, the drones try to avoid it more actively, often making the swarm easily stuck or stalled.

Tuning the Integrated Fix for A1. To make the integrated fix work, we tuned the fix. Specifically, when we combine the individual fixes, we tune the fix for C1-2 and C1-6. The original fixes for C1-2 and C1-6 add `0.15` and `200` to `influence_radius` and `repulsive_coef`, respectively. We reduce the increment in half: `0.075` and `100`, resulting in the final value of `0.225` (originally `0.15`) and `300` (originally `200`) for `influence_radius` and `repulsive_coef`, respectively. With the tuned fix, SWARMFLAWFINDER was not able to find logic flaws for 24 hours.

Fixes for Others. For A1~A4, all individual fixes successfully resolve targeted logic flaws. The integrated fixes for A2 and A3 resolved all the logic flaws. For A4, we observe conflicting fixes when we integrate the fixes. Details can be found in [8].

E. Influence of Moving Obstacles to our Evaluation

In our evaluation (§ V), A1's mission contains a moving obstacle. To understand its impact on our experiment results, we run the experiments again *without the moving obstacle*. Table VI shows the result. While there are small differences in the number of executions, the number of unique mission failures is mostly identical except for 4 flaws in C1-1 and C1-2 (marked as yellow and red cells). Those four missing unique mission failures are either directly caused by the obstacle (i.e., crashed into the obstacle; red cells) or indirectly caused (e.g., pushed by the dynamic obstacle leading to a crash to other drones; yellow cells).

F. Root Causes and Potential Fixes

C1-3. *Unsupported static movement:* A1 and A4 do not allow a drone's static movement, meaning that a drone has to move on every tick, even if it is desirable to maintain the same pose. The design of the algorithms does not consider the static movement, causing crashes in a crowded area.

> *Fix (Confirmed):* We change the constraints that make drones always moving (8 SLOC).

TABLE VI
INFLUENCE OF MOVING (OR DYNAMIC) OBSTACLES

ID	Root Cause	With Dyn. Obj.		Without Dyn. Obj.	
		# of Exec.	Uniq.	# of Exec.	Uniq.
A1	**Crash between Victim Drones**	**273**	**9**	**223**	**7**
	C1-1	86	4	78	3
	C1-2	176	4	132	3
	C1-3	11	1	13	1
	Crash into external objects	**435**	**8**	**378**	**6**
	C1-1	88	3	53	2
	C1-2	326	3	297	2
	C1-3	3	1	5	1
	C1-4	18	1	23	1
	Suspended progress	**671**	**2**	**622**	**2**
	C1-5	242	1	231	1
	C1-6	429	1	391	1
	Slow progress	**175**	**1**	**181**	**1**
	C1-6	175	1	181	1

C1-6, C3-2, and C4-3. *Insensitive object detection:* A victim drone's sensitivity in detecting objects is too low, making the entire swarm less reactive and sluggish in reacting to external objects and attack drones. We observe that a single attack drone can slow down the entire swarm due to this.

> *Fix (Confirmed for [4], [6]):* We change `repulsive_coef`, `sensing_radius`, and `IR_dist` configuration variables with the values of 400, 10, and 4 respectively. The developers of [4], [6] agreed with our analysis and the fix.

G. Supporting a new Algorithm

Our design is general and applicable to other swarm algorithms while it requires engineering effort. To support a new swarm algorithm, we need to instrument the algorithm to integrate SWARMFLAWFINDER (e.g., changing 218, 271, 198, and 166 SLOC for A1, A2, A3, and A4, respectively). In our evaluation, it took 8~15 hours (by a graduate student with moderate experience in drones) to complete this task for an algorithm. Details including the additional code are on [8].