TOWARD ROBUST SWARM ALGORITHMS VIA PRECISE CAUSAL ANALYSIS

Chijung Jung

Charlottesville, Virginia

A Dissertation submitted to the Graduate Faculty

of the University of Virginia in Candidacy for the Degree of

Doctor of Philosophy

Department of Computer Science

University of Virginia

May 2024

Yixin Sun, Chair

Yonghwi Kwon, Advisor

Tianhao Wang, Member

Kyusang Lee, Member

Kyu Hyung Lee, Member

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

Chijung Jung

This dissertation has been read and approved by the Examining Committee:

Yixin Sun, Committee Chair

Yonghwi Kwon, Advisor

Tianhao Wang, Committee Member

Kyusang Lee, Committee Member

Kyu Hyung Lee, Committee Member

Accepted for the School of Engineering and Applied Science:

Jennifer L. West, School of Engineering and Applied Science

Acknowledgments

Thank you to Professor Yonghwi Kwon, my advisor, who has guided me in the right direction for my Ph.D. study. You have helped me a lot to become a better researcher. I learned countless good aspects of him, such as how to write a good paper, how to organize thoughts in a logical way, and how to make the process systematic, which are the basic but the most important attitudes that a Ph.D. student has to have. I am extremely thankful to have worked with you for many years.

Also, thank you to my committee, Professor Yixin Sun, Tianhao Wang, Kyusang Lee, and Kyu Hyung Lee, for taking precious time out of their busy schedules to meet, for reading through this dissertation, and for the valuable comments/feedback they have provided.

Thank you to my friends in my research group. You guys are great collaborators, nice counselors, and comrades in arms who share the same trench. Special thanks to Ali, Jiho, Bora, Hamza, and all of the other members of the R3S Lab at UVA. My work has become better because of your help.

Thank you to my friends, Byeongu Jin, Seokhwan Song, Kyui Jeon, Hyungjin Kim, Dr. Seunghwan Ryu, Dr. Junhwan Han, my counselor, Katie Fowler, and the people in Korean Community Church in Charlottesville. I could not have maintained my mental health without you. Thank you to my friends and colleagues in ROKA, Seungwon Baik, Sangsu Kim, Professor Jonghwan Kim, and Youngmin Kim for your constant support. Also, special thanks to Col. Kangkyun Jin, Gen. Yonghyo Kim, and Col. Heewon Yang for giving me this opportunity to pursue the degree.

Thank you to my family for all the support you have given me throughout my Ph.D. student period and my life. Mom, Dad, and sister, thank you for the love and support you've given throughout my life. Also, thank my mother-in-law, father-in-law, and grandma, Dorcas. Whenever I need it, you always give me a lot of help, even in the U.S.

Finally, I thank my family. My wife, Hyunmyung, is a hidden collaborator for all of my work. This work would not have been possible without her sacrifice for the family. Also, My kids Taeyang, Belle, and Luke are my infinite source of happiness. I love you all.

Abstract

Swarm robotics is an emerging research area due to its diverse applications, such as environmental monitoring, disaster recovery, logistics, and even military operations, which are challenging for individual robots. Under the hood, a swarm algorithm is the core decision-making component that controls and coordinates multiple drones. Testing a swarm algorithm is crucial for developing robust drone swarms. However, it is challenging to analyze swarm systems due to the overwhelming complexity and dependencies among the components. Swarm is highly reactive to various environmental factors (e.g., obstacles), and swarm algorithms make extremely dynamic decisions based on them. In particular, swarm behavior is difficult to measure, which is critical for understanding swarm algorithms. Unfortunately, existing metrics (e.g., swarm size, coherence, or accuracy) have a limited reflection of dynamic behavior change caused by the impact of environmental factors. In this work, we propose systematic approaches that debug configuration bugs, discover logical flaws, and generate tests for swarm algorithms. In particular, we introduce a novel abstraction of robotics behavior, which we call the degree of causal contribution (Dcc), based on the idea of counterfactual causality. By leveraging Dcc, we measure swarm behavior in terms of interaction with environmental factors. First, we propose a swarm debugging system that automatically diagnoses and fixes buggy behaviors caused by misconfiguration. Then, we build a feedback-guided greybox fuzz testing system to discover logic flaws, leveraging Dcc as a feedback metric. We also build a system that generates tests with an enhanced mission environment so that the swarm leads to more complex behavior. We evaluate our approaches using real-world swarm algorithms to show generality and effectiveness. We also conduct real-world experiments using physical drones to show their applicability in the real-world.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Inspired by swarms in nature, swarm robotics revolutionizes how robots can function and what they can accomplish. It has attracted attention for a variety of vital missions, such as environmental monitoring, disaster recovery, logistics, and even military operations, 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.

The core of the swarm operation is the swarm algorithms that allow individual robots of the swarm to plan, share, and coordinate their trajectories and tasks to achieve a common goal. However, developing robust swarm algorithms is challenging. (1) Swarm algorithms are dependent on a large number of related parameters and inputs that can significantly change the behavior of the swarms. The swarm algorithm controls multiple robots adding an order of magnitude in complexity to a large number of parameters used to configure each robot. (2) Swarm operations are highly dynamic, compounding the variability and sensitivity of all its robots to the environment. (3) Swarm algorithms have variables and code blocks that are highly inter-dependent. The algorithms are often a closed-loop (feedback) control system [63, 118] which continuously computes robots' new states using new inputs and their previous states.

We present three approaches for robust swarm algorithms in this work. First, we propose a SwarmBug, a swarm debugging system that automatically diagnoses and fixes buggy behaviors caused by misconfiguration. Swarm algorithms rely on a large number of configurable parameters that can be tailored to target particular scenarios. This large configuration space, the complexity of the algorithms, and the dependencies with the robots' setup and performance make debugging and

fixing swarm configuration bugs extremely challenging. As a result, one of the common challenges in swarm algorithms and robotics development is to find appropriate values for configurable parameters. Even when the swarm algorithm has no logic flaws, a slightly misconfigured parameter can cause a buggy behavior, which we call *configuration bugs*. This work addresses *configuration bugs* in swarm algorithms (i.e., bugs caused by misconfiguration of the algorithms and robots), causing incorrect swarm states (such as crashing drones) in a particular deployment scenario.

A typical debugging approach for a configuration bug might be tracking each parameter's value propagation to the robot's decision that caused a faulty scenario. Unfortunately, the complexity of swarm algorithms makes this approach impractical. Another typical approach is trial-and-error. A developer inspects a particular variable's value, modifies its value, and tests whether it will fix the bug. The debugging process typically requires non-trivial manual effort due to many configurable parameters and complex dependencies. Moreover, even after the developer identifies a potential fix (i.e., a new value for a configurable parameter), testing the fix in various scenarios is time-consuming and challenging due to the large space of possible swarm behaviors.

To this end, SWARMBUG targets bugs that are caused by misconfiguration of parameters (i.e., configuration variables). It aims to (1) find key variables that caused a buggy behavior, (2) identify possible fixes for the bug via systematic testing, and (3) rank the fixes that preserve the behavior of the original execution. The essence of SWARMBUG is the novel concept called the degree of causal contribution (DCC), which abstracts impacts of environment configurations (e.g., obstacles) to the drones in a swarm via behavior causal analysis. SWARMBUG leverages DCC to understand which factors are causally contributing to the buggy behavior. SWARMBUG then finds variables that can configure swarm algorithms to adjust the DCC of the factors.

SWARMBUG automatically generates, validates, and ranks fixes for configuration bugs. We evaluate SWARMBUG on four diverse swarm algorithms. SWARMBUG successfully fixes four configuration bugs in the evaluated algorithms, showing that it is generic and effective. We also conduct a real-world experiment with physical drones to show the SWARMBUG 's fix is effective in the real-world.

Next, for detecting logic flaws in swarm algorithms, particularly in *drone swarms*, we propose a sys-

tematic approach. Specifically, we develop an automated testing system, called SwarmFlawFinder. We identify and overcome various challenges in understanding and reasoning about the swarm algorithm execution. A key component is to design an efficient metric that abstracts a given test's effectiveness. Unfortunately, unlike testing traditional software [116, 190, 188], 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.

SwarmFlawFinder also introduces attack drones that 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). 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.

In addition, as identifying possible corner cases and unexpected scenarios becomes more important, the demand for a tool for constructing complex scenarios is increasing [113]. While autonomous robotics systems typically provide diverse testing scenarios by default, they are not complex enough to observe buggy behaviors because they focus on the demonstration of basic features. To construct

complex environments, the domain knowledge for the proper guidance that understand the interaction between swarm and environment is required. It also requires a lot of manual effort because it is challenging to measure the impact of the environment on the swarm's behavior. For this reason, the robotics testing community requires more tool support that can generate operating scenarios in a simulation environment automatically or semi-automatically [113].

To this end, measuring the complexity of the environment is critical to generating a more complex mission environment automatically. Unfortunately, existing metrics, such as accuracy [5, 178] or coherence [185, 25], are not effective in reflecting the complexity of the environment because they cannot infer the causality between obstacles and swarm's behaviors. We propose SwarmGen, a systematic approach for generating testing scenarios by mutating the environment for a swarm. Specifically, we develop a system that mutates obstacles in a way that makes it more challenging for the swarm by leveraging Dcc to measure the complexity of the mission environment.

We evaluate the proposed approach with four real-world swarm algorithms and ten missions. We test the missions using the mutated environments that are generated by SwarmGen and they discover 44 more unique behaviors and 13 more bugs than the original default environment.

Dissertation Statement. To develop robust swarm algorithms, finding and debugging bugs can be achieved by tests leveraging the causal analysis of swarm behaviors, which is abstracted by the metric.

Contributions. This work makes several novel contributions toward testing for the robust swarm algorithms:

- A novel concept, the degree of causal contribution (Dcc) that abstracts impacts of environmental factors (e.g., obstacles) to the swarm via behavior causal analysis. To build Dcc, multiple perturbed executions based on the idea of counterfactual causality are created to infer the causality between environmental factors and drones's behaviors.
- We develop a swarm debugging system, Swarmbug, that automatically diagnoses and fixes buggy behaviors caused by misconfiguration. We leverage the concept of Dcc to understand

the degree of causal contribution of each variable to swarm behavior and use it to precisely pinpoint critical variables that contribute to bugs. We evaluate our algorithm on 4 real-world swarm algorithms and automatically identified 7 valid bug fixes, including physical flight experiments with real-world drones to empirically show that the generated fixes are effective in real-world scenarios. Also, we have communicated and confirmed all the configuration bugs and our fixes with the authors of the swarm algorithms.

- We develop a greybox fuzz testing system for drone swarm 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 propose SwarmGen, a swarm environments mutation system, to find corner cases effectively. It leverages Dcc as a metric to measure the complexity of the mission environment, which is reflected in the swarm's behavior that interacts with the environmental factors. Testing using the mutated environments that are generated by the proposed approach discovers 44 more unique behaviors and 13 more bugs than the original default environments in 10 missions of 4 real-world algorithms.
- We publicly release all the developed tools, data, and results, including the prototypes of Swarmbug, SwarmFlawFinder, and SwarmGen, for the community.

Dissertation Organization. The remainder of this dissertation is organized as follows. In Chapter 2, we introduce Swarmbug, a swarm debugging system that automatically diagnoses and fixes configuration bugs. We present Dcc and the approach that observes the impact of changes in configurations on swarm behavior by leveraging Dcc in the section. We propose SwarmFlawFinder that discovers logical flaws of swarm algorithms in Chapter 3. To perturb a swarm, attack drones are introduced and we explain Dcc based fuzz testing approach with attack strategies in this section. In Chapter 4, we present SwarmGen, a mission environment generation technique for swarm testing, which causes more corner cases. The complexity of the mission environment, which is

reflected as the interaction between a swarm and environment in Dcc, is evaluated in the mutating process. Each section includes a background and summary subsection. In Chapter 5, we conclude and discuss possible future research directions.

Chapter 2

SWARMBUG: Debugging Configuration Bugs

2.1 Introduction

In robotics, a swarm is a group of cooperative robots that is able to solve complex tasks through their collective behavior [51]. Swarms are being used to solve many real-world problems, from environmental monitoring and emergency response to entertainment [155]. Key enablers of such success are the algorithms that allow the individual robots of the swarm to plan, share, and coordinate their trajectories and tasks to achieve a common goal [37].

Despite the potential of swarms, developing robust swarm algorithms is challenging. (1) Swarm algorithms are dependent on a large number of related parameters and inputs that can significantly change the behavior of the swarms. The swarm algorithm controls multiple robots adding an order of magnitude in complexity to a large number of parameters used to configure each robot (e.g., ArduCopter [10] has hundreds of configuration parameters). (2) Swarm operations are highly dynamic, compounding the variability and sensitivity of all its robots to the environment. (3) Swarm algorithms have variables and code blocks that are highly inter-dependent. The algorithms are often a closed-loop (feedback) control system [63, 118] which continuously computes robots' new states using new inputs and their previous states.

In our conversation with developers of swarm algorithms [2, 164] and observation from public forums [168, 167, 191, 94, 108], one of the common challenges in swarm algorithms and robotics

development is to find appropriate values for configurable parameters. A slightly misconfigured parameter can cause a buggy behavior, which we call *configuration bugs*. This research focuses on *configuration bugs* in swarm algorithms (i.e., bugs caused by misconfiguration of the algorithms and robots), causing incorrect swarm states (such as crashing drones) in a particular deployment scenario.

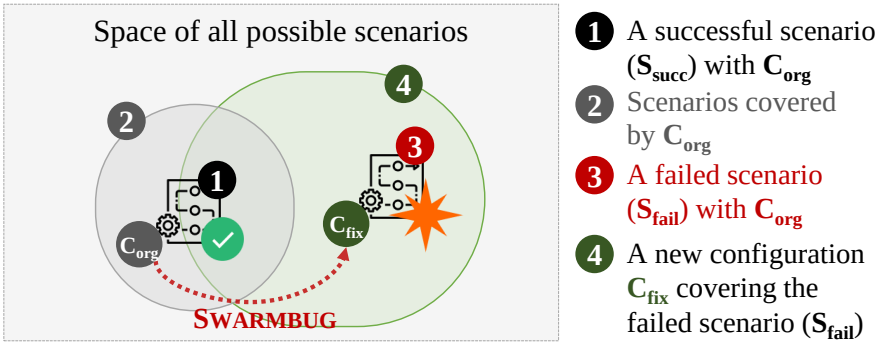


Figure 2.1: Illustration of a configuration bug and SWARMBUG.

Configuration Bugs. Figure 2.1 illustrates a high-level concept of the configuration bug and the SWARMBUG's ultimate objective. Given the space of all possible scenarios (S_{all}) of a swarm, there is a configuration for the swarm (C_{org}) that can result in a successful scenario (S_{succ}) denoted by ❶. ❷ denotes scenarios that can be successfully covered by C_{org}. A configuration bug happens when a swarm operates under a new scenario resulting in a failure S_{fail} because it is not covered by C_{org}.

Challenges. A typical debugging approach for a configuration bug might be tracking each parameter's value propagation to the robot's decision that caused a faulty scenario. Unfortunately, the aforementioned complexity of swarm algorithms makes this approach impractical. For example, parameters often go through a number of complex computations with other variables, including matrix multiplications. Precisely tracking a variable's impact after those computations is an extremely challenging task. Another typical approach is trial-and-error. A developer inspects a particular variable's value, modifies its value, and tests whether it will fix the bug. The debugging process typically requires non-trivial manual effort due to many configurable parameters and complex dependencies. Without proper guidance on each trial-and-error, this approach is rather impractical.

Moreover, even after the developer identifies a potential fix (i.e., a new value for a configurable parameter), testing the fix in various scenarios is time-consuming and challenging due to the large space of possible swarm behaviors.

Our Approach. This research proposes SWARMBUG, a swarm debugging approach for configuration bugs. As illustrated in Figure 2.1, it aims to find a new configuration which we call a fix C_{fix} that can cover more scenarios (4). While not guaranteed, SWARMBUG prioritizes C_{fix} that are close to the C_{org}, which can potentially cover some of the scenarios already covered by C_{org} (2) (as per the overlapping area of 2 and 4).

In particular, SWARMBUG targets bugs that are caused by misconfiguration of the swarm algorithm or robot's parameters (i.e., configuration variables). It aims to (1) find key variables that caused a buggy behavior, (2) identify possible fixes for the bug via systematic testing, and (3) rank the fixes that preserve the behavior of the original execution. SWARMBUG's key enabling technique is the novel concept of the degree of causal contribution (DCC). It creates alternative executions with and without critical factors (e.g., objects) that affect the swarm's behavior to understand which factors are causally contributing to the buggy behavior. SWARMBUG then finds variables that can configure swarm algorithms to adjust the DCC of the factors. The contributions of this research are as follows:

- We develop a swarm robotics debugger for configuration bugs.
- We propose the concept of DCC to understand the degree of causal contribution of each variable to swarm behavior and use it to precisely pinpoint critical variables that contribute to bugs.
- We evaluate our algorithm on 4 real-world swarm algorithms and automatically identified 7 valid bug fixes, including physical flight experiments with real-world drones to empirically show that the generated fixes are effective in real-world scenarios.
- We have communicated and confirmed all the configuration bugs and our fixes with the authors of the swarm algorithms.
- We publicly release the source code and data of SWARMBUG on *https://github.com/swarmbug/src*.

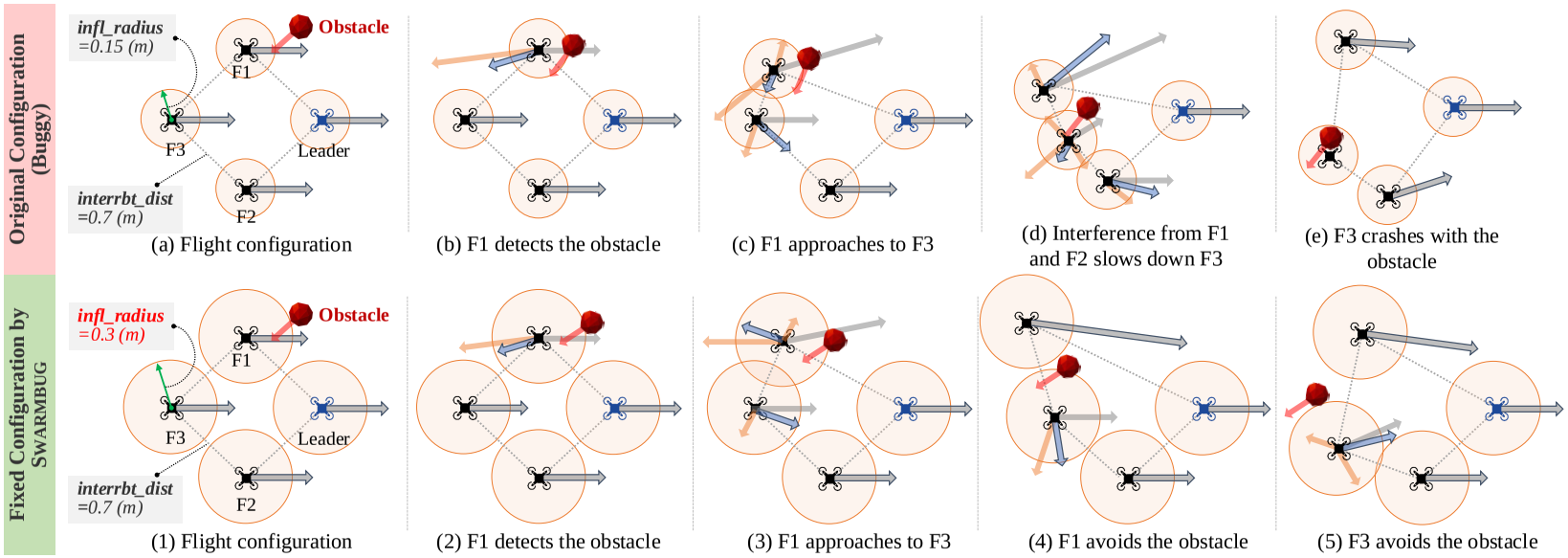


Figure 2.2: Swarm of four drones crashing an obstacle: (a)∼(e). The same swarm mission with a fix by SWARMBUG: (1)∼(5).

2.2 Motivating Example

We use the Adaptive Swarm [2] algorithm to illustrate SWARMBUG's operation. We run the algorithm for four drones: one leader and three follower drones (F1∼F3). The algorithm's goal is to safely move the swarm to a destination while maintaining a diamond-shape formation as shown in Figure 2.2-(a). The arrows with borders (either blue or gray) indicate the drone's flight direction. Orange arrows are the vectors caused to avoid obstacles (including other drones). Gray arrows represent the vector to maintain the diamond formation. When there are multiple vectors considered, the blue arrows with borders indicate the final flight directions.

Configuration Variables. In this example, there are two types of configuration variables: environment and swarm configuration variables. Environment configuration variables represent objects such as robots and obstacles (e.g., *followers[0∼1].sp*, *self.sp.x*, and *obstacle[8]* in Figure 2.3). Swarm configuration variables are parameters for swarm algorithm and robots. For example, circles surrounding drones visualize a parameter *infl_radius* that determines the maximum sensing distance for objects. *interrbt_dist* is another parameter that represents the desired distance between drones.

Configuration Bug. Figure 2.2-(b)∼(e) show such a scenario where F3 crashes with an obstacle due to a *configuration bug*. First, the *moving obstacle* approaches F1, which is also moving, in (b) and makes F1 move towards the south-west, leading F1 to get close to F3. In (c), the obstacle forces F1 and F3 closer. In (d), the obstacle approaches now F3 which fails to avoid it because the other four forces come into play: three forces to avoid F1, F2, and obstacles (oranges), and the force to maintain the formation. This causes F3 to move just slightly from its current position, not enough to avoid the obstacle, leading to a crash in (e). A cause for the failure is that, in (d), F3 was too close to adjacent drones which interfere with the decision of F3 to avoid the obstacle.

Debugging Attempts without SWARMBUG. A typical debugging approach of the given bug is to trace the value propagation from the obstacle (i.e., the cause of the crash) to the drone to understand how the obstacle and other variables affect the drone's faulty decision. For example, one may use existing program analysis techniques such as taint analysis [40, 85, 12, 144, 163]

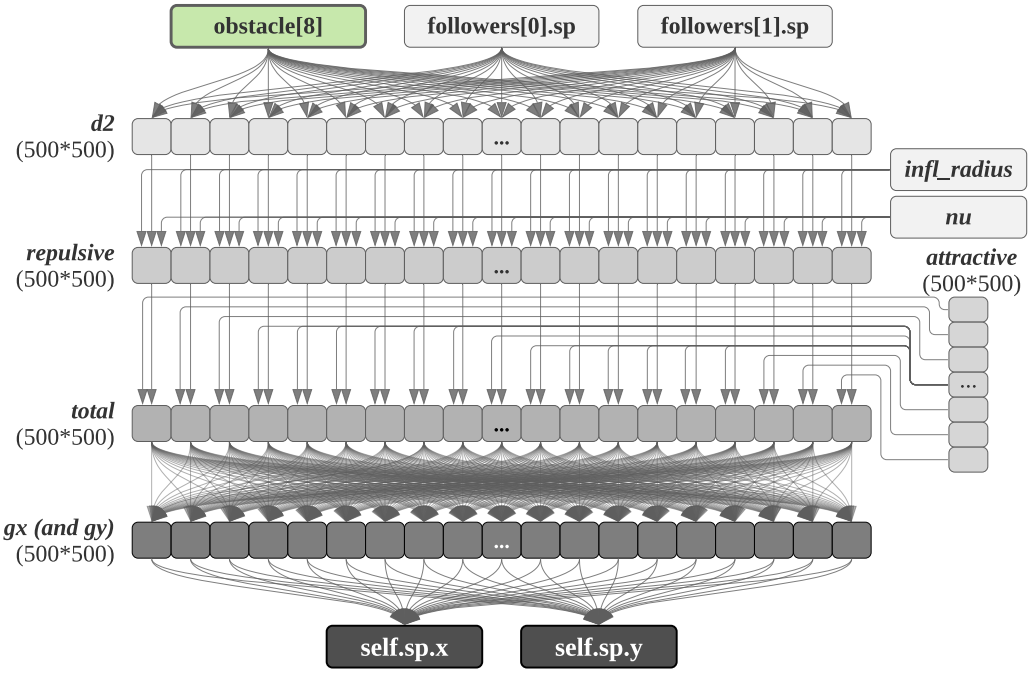


Figure 2.3: Illustration of simplified value propagation.

to trace *obstacle[8]* which is an environment configuration variable (defined as a global variable) representing the obstacle. Each drone in the swarm reads this variable to determine whether they are close to the moving obstacle or not. However, tracking the value propagation of the variable is challenging as it goes through complex computations.

Figure 2.3 shows a simplified value propagation graph. The arrows in Figure 2.3 show the data propagation paths. The source variable (*obstacle[8]*) is a 2×4 array and the values of its elements (along with other variables including *followers[0].sp* and *followers[1].sp* representing other drones) are used to generate each element of a 500×500 array, *d2*. Later, each element of *d2* is used to create another 500×500 array *repulsive* with *infl_radius* and *nu*. Then, each element of *repulsive* and *attractive* are added to create *total* (a 500×500 array). Finally, it computes a gradient of the matrix to create *gx* and *gy*. Finally, mean values of the gx and gy arrays to compute x (*self.sp.x*) and y (*self.sp.y*) coordinates. At this point, which part (of bytes) of the x and y coordinates are affected by the source variable *obstacle[8]* is challenging to know. Using taint analysis would tell

that every part of both coordinates depends on the source variable and other variables, which are not useful for debugging the configuration bug. Note that the graph is simplified. The complete graph of the swarm algorithm [2] is at least 10 times larger than Figure 2.3. A backward edge from the *self.sp.x* and *self.sp.y* to *followers[0].sp* and *followers[1].sp*, that forms cycles, are omitted.

Debugging with SWARMBUG. SWARMBUG (1) conducts a behavior causal analysis to find out environment configuration variables that caused the bug, (2) obtains bug fixes by mutating swarm configuration variables, and (3) ranks fixes that preserve the original behavior of the swarm.

(1) Cause Analysis: Given a definition of configuration variables provided by a user, SWARMBUG infers which configuration variables significantly contribute to the buggy behavior by leveraging a concept we call the *degree of causal contribution* (or DCC, details in Section 2.4.1). DCC essentially abstracts the impact (or contribution) of individual variables to a robot's decision.

DCC is computed as follows. Given the original execution (demonstrating the crash), SWARMBUG creates alternative executions by removing the impact of environment configuration variables (that are essentially related to surrounding objects and robots). Then, we compare the robots' behaviors of the original execution and the alternative executions. The difference of the robots' poses becomes a DCC value. Finally, we analyze the trends of DCC values around the time when the bug occurred to pinpoint the cause of the bug (e.g., whether some variable's contribution is insufficient or excessive). Note that SWARMBUG does not rely on tracking complex propagations of values, which existing techniques struggle to do, but rather analyzes values related to the robots' behavior as the environment is changed.

In the earlier example, SWARMBUG derives *alternative executions without each obstacle* by mutating environment configuration variables, to infer the causal relationship between an obstacle and the buggy behavior. Then, we compare each drone's poses observed during the generated alternative executions and the original execution, obtaining the difference that represents the impact of each removed obstacle to the buggy behavior. To this end, SWARMBUG identifies the most impactful variable: *obstacles[8]* (a moving obstacle).

(2) Finding Potential Configuration Fixes: From the environment configuration variable that contributes to the bug, SwarmBug conducts a number of experiments that change each *swarm configuration* variable's value (e.g., a robot's parameter's value) to identify potential fixes for the bug. Specifically, it focuses on the trend of Dcc values of the environment configuration variable. For example, we earlier noticed that the obstacle's contribution becomes more significant near the crash while other objects (e.g., other drones) also compete for the contribution.

To this end, SwarmBug tries to *reinforce* (or intensify) the increasing trend of the moving obstacle's Dcc value. With the change, we expect the drone to take the obstacle into account more significantly than the original execution. We then run multiple executions with mutated swarm configuration variables (e.g., increasing/decreasing their values) to find mutations that can reinforce the trend. Finally, we find concrete values for two swarm configuration variables (defined as global variables), leading to *two configuration fixes*: (i) *infl_radius*=0.3 and (ii) *interrbt_dist*=1.4.

(3) Validating the Robustness of Fixes: SwarmBug tests the two fixes (i.e., *infl_radius* and *interrbt_dist*) exhaustively, by running a number of tests with diverse scenarios that SwarmBug derived by profiling the variation of the target scenario (e.g., spawning the swarm in various positions). To make each test more meaningful in terms of validating the robustness, SwarmBug measures whether each run exercises observable new swarm behaviors using Dcc values. Specifically, for each test, we collect Dcc values and compute MSE scores against previous executions' Dcc values. The testing is repeated until it does not observe new swarm behaviors (e.g., MSE scores of 100 consecutive executions are all smaller than 0.01) or reached a predefined timeout (e.g., 20 hours). In this example, both fixes successfully pass the testing, meaning that SwarmBug did not observe any failures after 20 hours of testing while the fixes with *infl_radius* and *interrbt_dist* successfully finishes 3,880 and 1,211 tests respectively. Hence, both are considered as valid fixes.

(4) Finding Behavior-preserving Fixes: Some fixes may *disruptively* change the swarm behavior. For instance, in our example, changing *interrbt_dist* results in a bigger diamond formation, making the swarm look and behave quite differently. To avoid such fixes, SwarmBug aims to identify a *behavior-preserving* fix which behaves similar to the original swarm. Specifically, we compare

the Dcc values from a fixed execution and the original execution to measure the differences between the two executions. If two swarm executions have similar Dcc values, we consider that their behaviors are similar. In our example, the Dcc values from the fix with *infl_radius* is more similar to the Dcc values from the original run than the fix with *interrbt_dist*.

Chosen Fix: Figure 2.2-(1)∼(5) show the flight with the *infl_radius* fix. It maintains the same formation, while individual drone detects and avoids the obstacle earlier, preventing the situation where multiple drones get too close (2)∼(4). All the drones, including F3, avoid the obstacle successfully (5).

2.3 Backgrounds, goals, and scope

2.3.1 Mobile Robot Software

Configurable Variables. A typical robot such as the drones we use in our studies can have hundreds of configurable parameters and each of the parameters can affect the robot's behavior significantly. A robot's decision-making process is typically implemented as a sequence of program statements that *continuously and iteratively* reads inputs from various sensors and computes the robot's next state, meaning that it is essentially a closed-loop system [204]. During the computation, the configurable parameters are also taken into account. As shown in Figure 2.3, variables in the algorithms are highly inter-dependent (e.g., most variables in the loop are dependent on their previous iteration's values), making it difficult to apply data-dependency analysis techniques.

Field Testing and Simulation-based Testing. Testing robotics algorithms is challenging because robots interact with the physical surroundings. While testing robots in the real-world (field testing or physical testing) is desirable and ultimately required, it is expensive and dangerous due to the cost of failures. As a result, simulation-based testing is a common alternative that can reduce development and validation costs. Still, given the dimension and complexity of the real-world, simulation-testing must identify what scenarios are worth validating and attempt to reduce the

exploration of equivalent scenarios that render little value for testing.

2.3.2 Swarm Algorithms

Centralized and Distributed Swarm Algorithms. There are two main lines in constructing swarm algorithms [124, 81, 37, 16, 13]: centralized and distributed. A centralized algorithm [121, 46, 27] computes all the decisions of individual robots in a swarm in a centralized system. On the other extreme, a distributed swarm algorithm [98, 11, 194] runs the majority of the algorithm on individual robots, where robots are communicating via network channels. Existing approaches such as taint analysis have difficulty handling distributed algorithms while SWARMBUG works well on both centralized and distributed algorithms.

Local vs Global Goals. Swarm algorithms may have global goals for the entire swarm and local goals for individual robots at the same time, leading to *conflicting goals*. For instance, each robot may have a local algorithm to avoid obstacles, while a swarm algorithm aims to maintain a specific formation during the flight. When a robot in the swarm encounters an obstacle, the robot's local algorithm may hold back the swarm algorithm's progress as it prioritizes its local goal (i.e., avoiding the obstacle). Note that even if a swarm algorithm includes logic to balance the two goals (e.g., prioritizing local and global goals based on the current state and environment), the balancing logic may not be perfect, failing to balance the conflicting goals.

Complex Dependencies. As a swarm consists of multiple robots, the complexity of dependencies among variables and configurations has significantly increased compared to that of a single robot. During our experiments, we observe that the average number of data dependencies (i.e., the number of edges in the data dependence graph) in drone swarm algorithms [110, 109, 80, 134, 193] is '1,693+1,207$*n$' where n represents the number of robots.[1] When n=5, the number is approximately 3.7 times the average number of dependencies of algorithms for a single drone which is 2,042 [130, 29, 56, 147, 52] (with n=10, the swarm algorithms' dependencies are 6.7 times bigger

[1] As for '1,693' and '1,207', we use the data-dependency graph using Sourcetrail [161], with T as the total edges of the swarm algorithm and L as the number of edges for an individual drone algorithm. '1,693' is the average of the difference between T and L, and '1,207' is the average of L of all drones.

than the single drone algorithms). It means that applying the data dependency analysis to swarm algorithms is ineffective in practice.

Dynamic Behaviors. In a swarm, individual robots' dynamic behaviors are often accumulated and amplified, leading to even more diverse swarm behaviors. For example, in our motivation example, Figure 2.2-(c) and (d) have a chain reaction to the obstacle, which is different from when an individual drone interacts with an obstacle. Hence, a significant challenge in swarm testing is obtaining test cases that can effectively cover various swarm behaviors and prioritizing test cases to cover diverse scenarios.

2.3.3 Goals and Scope

Goals of SWARMBUG. SWARMBUG aims to achieve the three major goals to effectively debug swarm algorithms as follows.

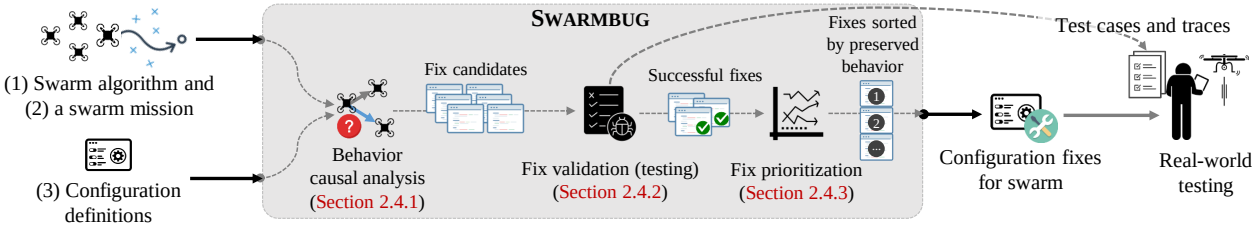


Figure 2.4: Overview of SWARMBUG

- **Goal-1**: *Developing effective causal analysis capabilities for swarm algorithms* to automatically identify root causes of configuration bugs and find fixes.
- **Goal-2**: *Developing an effective and efficient testing approach* to validate bug fixes for swarm algorithms by systematically covering various corner cases.
- **Goal-3**: *Understanding the impact of fixes and guiding how to choose fixes* that preserve the original swarm algorithm's behavior while correcting buggy behaviors.

Focus on Unmanned Air Vehicles (Drones). While our findings and insights are generic

and applicable to various swarm robotics environments, our research focuses on swarm robotics algorithms for unmanned aerial vehicles. This is because (1) they are prevalent and used in various missions, and (2) they have one of the most sophisticated dynamics, leading to various challenges in debugging.

Generality of SWARMBUG's Fix. SWARMBUG generates fixes for a bug under a particular mission and algorithm's configuration. This means that the fixes may not work for a significantly different mission or scenario. For instance, a bug fix for a swarm mission with four drones may not work for a mission with eight drones. Also, a bug fix for a swarm avoiding obstacles may not work if the obstacles' speed changes (e.g., become faster).

2.4 Design

Figure 2.4 shows the overall procedure of SWARMBUG. It takes three inputs: (1) a swarm algorithm's source code, (2) a swarm mission that triggers a buggy behavior, and (3) configuration definitions that include a list of configuration variables for the swarm and environment (e.g., certain obstacles, wind, etc.). SWARMBUG conducts a behavior causal analysis (Section 2.4.1) to find causes of buggy behaviors from environment configuration variables and generate fixes for swarm configuration variables. Then, SWARMBUG validates the fixes under various scenarios (Section 2.4.2) to obtain robust fixes. Further, it ranks the fixes based on the behavior similarity between the original swarm and the fixed swarm (Section 2.4.3). Finally, while it is not part of our main contribution, the test cases and traces can be used to conduct real-world testing as shown in Section 2.5.2.

2.4.1 Behavior Causal Analysis

Configuration Variables

Among the variables in a swarm algorithm, there are two types of variables that are important in understanding and controlling behavior: environment and swarm configuration variables. One of

the SWARMBUG's inputs is the configuration definitions: a list of configuration variables with each variable's type (either environment or swarm configuration) and the value specification.

1. **Environment Configuration Variables** define the environment of the swarm that can be manipulated during simulation such as obstacles, robots, and wind. The value specification includes a value to eliminate the impact of the variable. For instance, if an obstacle is defined as a set of coordinates, coordinate values outside of the map will effectively remove the obstacle. We use the ∅ symbol to represent such a value.
2. **Swarm Configuration Variables** typically define parameters of drones and swarm algorithms. The specification includes the range of values (i.e., minimum and maximum values, distribution). For instance, the maximum drone velocity or the minimum distances between drones in a swarm.

Profiling for the Configuration Definitions. SWARMBUG expects a user to provide the configuration definitions[2], which may require non-trivial effort. To mitigate this, we present a set of profiling tools and supporting approaches on our project website [166] and Section A.1 that can generate sketches of such configuration definitions for implementations like the ones we present later in our study [184, 2, 132, 72] to reduce such effort.

Degree of Causal Contribution (DCC)

Our analysis targets environment configuration variables that represent obstacles and other robots because *they directly affect the swarm behavior* and are crucial in understanding causes of bugs. A key innovation of SWARMBUG is the concept of *the degree of causal contribution (or* DCC*)* of a variable to a robot's pose and propose its computation without relying on complex data propagation analysis techniques such as taint analysis. DCC is computed by comparing differences between executions with mutations applied on the environment configuration variables.

Computing Delta (Δ) via Alternative Execution. To understand the contribution of an

[2]Details of the configuration definitions and the real input file we use in this research can be found on *https://github.com/swarmbug/src/tree/main/Input_Swarmbug*

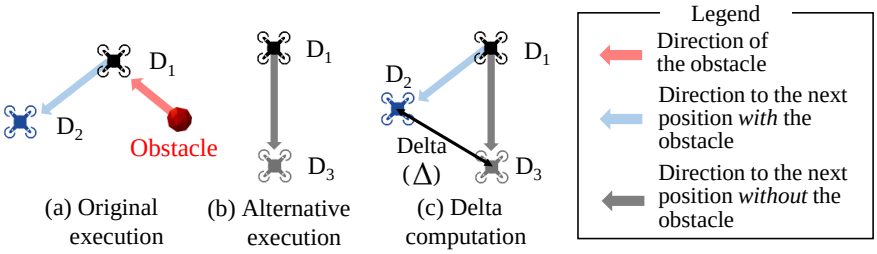


Figure 2.5: Example of computing a delta (Δ) value.

environment variable, we first create a new (alternative) execution with a mutation on the variable that can essentially remove the variable's presence in the environment. Since the new execution negates the existence of the mutated variable, we call the new execution *alternative execution*. Figure 2.5 shows an example. Suppose that Figure 2.5-(a) shows an original execution that includes an obstacle, leading to the drone moving toward the south-west (from D_1 to D_2). A counterfactual execution is shown in Figure 2.5-(b) without the obstacle. The drone moves toward the south (from D_1 to D_3). As shown in Figure 2.5-(c), we obtain a delta by computing the Euclidean distance between drones' poses (D_2 and D_3) from the two executions.

Computing Dcc. The degree of causal contribution (or Dcc) is an aggregation of the delta (Δ) values of environment configuration variables. Specifically, we obtain Δ values of all environment configuration variables. Then, we compute the percentages for each variable, resulting in Dcc.

Algorithm 1 shows the details of Dcc computation. Given a swarm algorithm, it iterates over all the robots in the swarm and calls *ComputeRobotDcc* for every tick to obtain all Dcc values in the given mission M (lines 1-6). Then, it obtains the robot's pose (i.e., coordinate) in the original mission at the given tick t by calling *GetRobotPose* and stores the results to P_{org} at line 9. We remove each environment configuration variable's impact (i.e., v_i) by assigning $\varnothing$ to v_i. Next, we obtain a new robot's pose (line 13) without the object v_i, and store it to P_i. We compute delta Δi by calculating Euclidean distance between P_{org} and P_i (line 14). We modify v_i's value on each iteration to remove the object (line 12), and restore it (line 16). Finally, we construct a set of proportions of individual variables' deltas (line 19).

Algorithm 1 Computing Dcc from the Delta values

Input : M: a set of missions for robots. $m_r \in M$ is a mission for robot r,
T_e: the tick value of when the swarm mission finishes.
V_{ec}: a set of environment configuration variables.
Output: Dcc(r, t): a set of tuples $<v_{ec}, N>$ where v_{ec} is an environment configuration variable and N is the Dcc value of v_w at tick t for robot r

```
1  procedure ComputeSwarmDcc(M, V_w)
2      t ← 0
3      while t ≠ T_e do
4          for m_r ∈ M do
5              Dcc(r, t) ← ComputeRobotDcc (m_r, V_ec, t)
6          t ← t+Time-step // Time-step represents a single tick
7  procedure ComputeRobotDcc(r, V_ec, t)
8      Δtotal ← 0
9      P_org = GetRobotPose (r, V_ec, t) // Obtain a pose of r at t
       // Each source variable v_i representing a world object
10     for v_i ∈ V_ec do
11         tmp ← v_i // Save v_i
12         v_i ← ∅ // Removing the impact of an environment configuration variable v_i
13         P_i = GetRobotPose (r, V_ec, t) // Obtain a pose of r at t without v_i
14         Δi ← ||P_org − P_i|| // Δ for v_i via Euclidean Distance
15         Δtotal ← Δtotal + Δi
16         v_i ← tmp // Restore v_i
17     dccSet ← {}
18     for v_i ∈ V_ec do
19         dccSet ← dccSet ∪ <v_i, (Δi / Δtotal)>
20     return dccSet
```

Temporal Analysis

We analyze how Dcc values change over time (i.e., trend) to identify the causes of a bug.

Time Window for Temporal Analysis. Robots typically have some lag in recognizing and reacting to changes in their surroundings. We call such time duration T_{win} (or time window for temporal analysis), and focus on the trend of Dcc values within the window. Note that different swarm algorithms may have different time windows so test missions are typically provided by the developers or can be obtained with slight changes of their configuration. Then, we identify when the current Dcc value is changed more than 10% than its previous tick's Dcc value (i.e., Dcc value is rapidly changing). Note that the 10% threshold is configurable[3].If such rapid changes are observed, we record how long the changing trend lasts. We calculate the average time they last and use it for the time window, T_{win}. In this research, we measured T_{win} values of 7.6 ticks, 100 ticks, 6 ticks, and

[3]The optimal for each algorithm can be profiled. Details can be found in [166] or Section A.3. We profile the four algorithms we evaluated, and find that 10% works for all of them.

3 ticks for Adaptive Swarm [2], Swarmlab [184], Fly-by-logic [132], and Howard's [72] respectively.

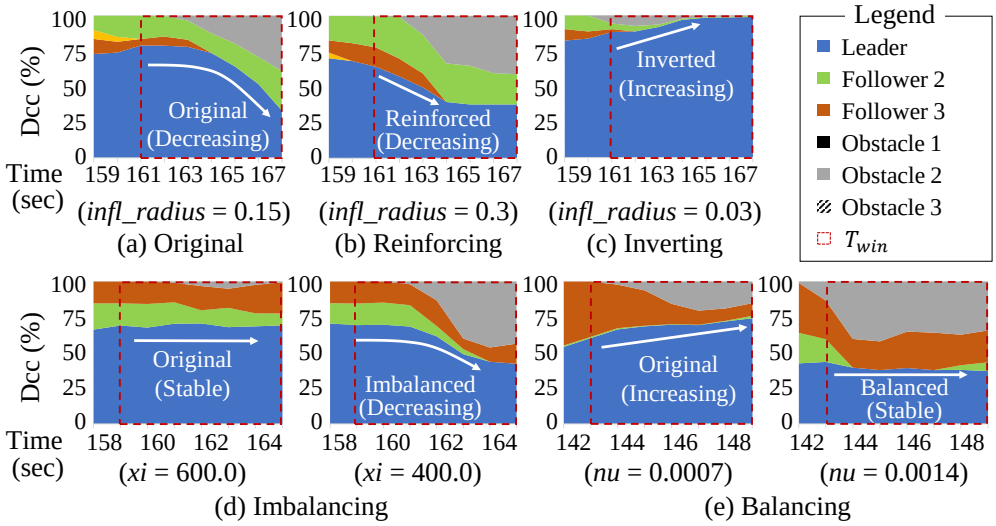


Figure 2.6: Examples of Dcc value trends and fixing strategies.

Fixing Strategies Based on Dcc Trends. With the identified T_{win}, we try to identify a temporal trend of Dcc values within a family of predefined templates, as shown in Figure 2.6, that reflect our experience in practice. Then, we apply a set of predefined fixing strategies depending on the matching temporal trend template. From the time that it causes a buggy behavior, T_{bug}, the time window for our temporal analysis starts at '$T_{bug} - T_{win}$' and ends at 'T_{bug}', as shown in Figure 2.6. Then, we apply the following four strategies.

1. **Reinforcing.** If the trend of Dcc values is either increasing or decreasing, we try to *reinforce* the trend (i.e., increasing or decreasing more). Figure 2.6-(a) shows an example of a decreasing trend of Dcc values. Figure 2.6-(b) is a fix obtained by changing the value of *infl_radius* (a swarm configuration variable that represents the maximum sensing distance for objects) to 0.3 from 0.15 (the original value shown in Figure 2.6-(a)).
2. **Inverting.** If Dcc values are increasing/decreasing, we generate a fix to invert (i.e., decrease/increase) the trend of Dcc values, respectively. For example, Figure 2.6-(c) inverts the trend of Dcc values from Figure 2.6-(a) by changing the value of *infl_radius* to 0.03 (from 0.15). This strategy is effective when a swarm overlooks an essential factor and focuses on trivial inputs. It

would invert the focus so that the essential factor can be considered.

3. **Imbalancing.** If a Dcc value of the variable does not have noticeable changes, we try to introduce changes that can lead to different swarm behavior. We first try to imbalance (i.e., either increase or decrease) the Dcc values.

   For example, Figure 2.6-(d) introduces a decreasing trend by changing the value of *xi* (a swarm configuration variable) to 400 from 600. *xi* represents the non-leader robot's tendency of following the leader drone. Reducing this value allows robots to focus on other surroundings.

4. **Balancing.** Swarm algorithms may fail because they accidentally take some inputs into the computation more or less than they should be. This strategy will try to reduce the impact of overly-prioritized objects in algorithms. For example, Figure 2.6-(e) changes the value of *nu* from 0.0007 to 0.0014. *nu* swarm configuration variable representing the priority of avoiding obstacles over other goals (e.g., following the leader). The fix prevents the drone from being overly considering the leader.

2.4.2 Fix Validation

Profiling Spatial Variations

It is common to observe a swarm behaves differently between each test. A robust fix should be tested under such diverse behaviors. To understand the variation of a given swarm algorithm, we profile the drone's poses from tests.

Aligning Spatial Coordinates. Spatial coordinates of the swarm can vary across the test runs. For example, two relatively identical flights can have different coordinates if the entire swarm's poses are shifted. To identify the variation of drones' poses in the swarm, it is necessary to align the drones' poses based on common coordinate system. Specifically, we set the spatial coordinates of the swarm on the drone that caused a bug (e.g., a crash). Other objects including other drones and obstacles are referenced accordingly.

Computing Spatial Variations. We run n sets of tests where each set includes N tests ($N = 10$

in this research), until we reach a fixed point of the spatial variation. We measure the spatial variation of the drones' poses from all the test runs on each test set. For measuring the spatial variation SV, we leverage the concept of circular/spherical error probable (CEP/SEP) [43] to identify the area that can include 90% of coordinates from the total tests.

On the i^{th} test set, we measure the spatial variation of the drones' poses (SV_i) from all the test runs executed at this point ($i * 10$ tests). We repeat the process until we observe SV_{i-1} and SV_i do not differ more than 5%. In general, we reach the fixed point with 10 test sets, meaning that we run 100 tests in total. Details can be found on [166] or Section A.2.

To this end, we obtain a map called **SVMAP** (Spatial Variation Map) that shows the aligned spatial variations of individual robots and objects. Figure 2.7 shows an example **SVMAP** obtained from Adaptive Swarm [2]. In the map, observed robots are presented as points. Solid contour lines indicate areas that are estimated as the same density. The contour lines represent areas that contain the sample's population from 10% to 90%, where the outmost area includes 90%, and each inner area has 10% less population.

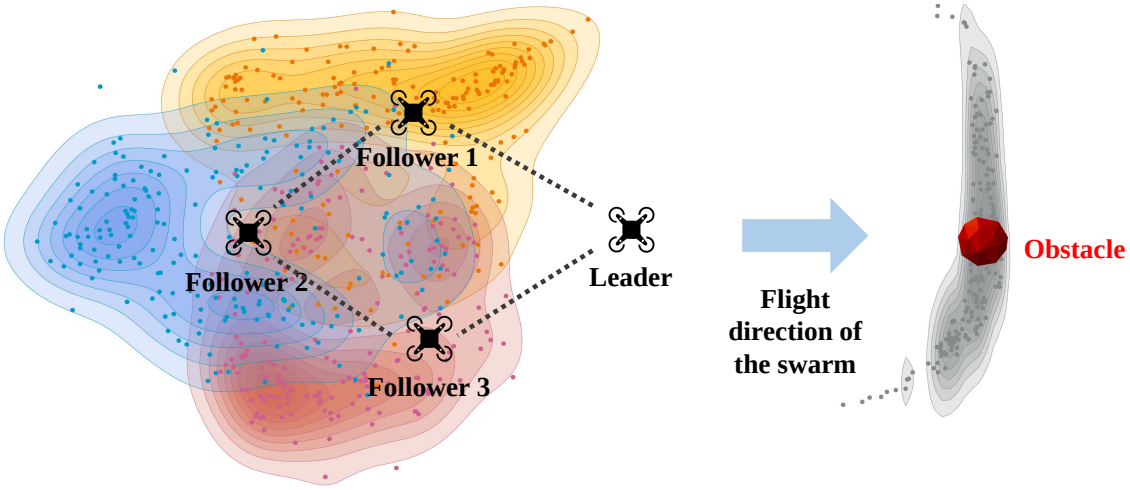


Figure 2.7: Spatial Variation Map (SVMAP)

Feedback-driven Fuzzing.

We validate the generated fixes by testing them under various scenarios. We use SVmap, which represents the spatial variation of the swarm under test. We aim to spawn robots and obstacles within the regions shown SVmap.

Initially, we spawn them in inner layers more than outer layers (because more drones were observed there during the profiling). During the tests, we record Dcc values. If the Dcc values of the current test differ by *more than 10% from all the previously observed* Dcc *values*, we consider the test covered some new swarm behaviors, hence a meaningful test covering a new scenario. In this case, we prioritize creating new tests that are similar to the current one. If the Dcc values from the current testing are similar to Dcc values from previous tests, we prioritize the other layers. Note that we essentially use Dcc values as feedback representing the behavior of the swarm. If we tried all the layers and cannot find new Dcc values that are more than 10% different from the previous tests, we extend the layers to cover larger spaces.

The process terminates (1) when the test fails (e.g., robots crashing to obstacles or walls) or (2) reaches a predefined timeout. If we reach the timeout without a failure, we consider the fix is valid. During the testing, if we observe any crashes or runs that fail to reach the original goal, we consider them unsuccessful runs, and the corresponding fixes are discarded.

2.4.3 Fix Prioritization

The fixes by Swarmbug may affect different aspects of the swarm behavior in an undesirable way. For example, a fix may resolve a crash by changing the swarm's formation significantly (increasing the distances between drones). In such a case, the swarm with the fix may look very different from the original one.

To this end, we *rank* the fixes by how much they preserve the original algorithm's behavior. Specifically, for each fix, we compare Dcc values from the swarm with the original configuration and fixed configuration. Then, we rank the fixes with smaller differences higher because they preserve

the original behavior of the swarm more than those with larger differences in Dcc values. In many cases, a higher-quality fix does not significantly change the swarm's behavior while eliminating the fault bug. Note that we essentially use Dcc to approximate the swarm behavior.

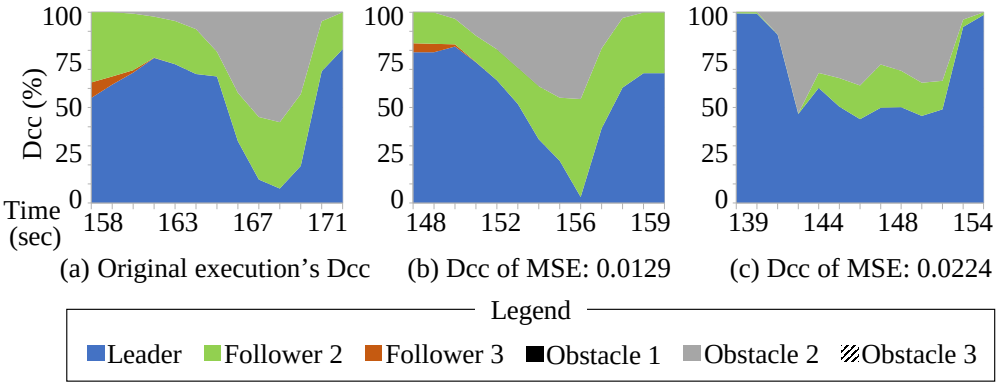


Figure 2.8: Example MSE scores

Measuring Distances of Dcc Values. To compare Dcc values from different runs, we leverage the Mean Squared Error (MSE). Figure 2.8-(a) shows Dcc values from the original execution, and Figure 2.8-(b) and (c) are Dcc values from executions with two different fixes. We rank the one with a smaller MSE value (0.0129) higher than the other with MSE value 0.0224. Note that we make them have the same length using interpolation (i.e., applying linear interpolation to the shorter sequence), then calculate the MSE to handle Dcc values of different time periods.

2.5 Evaluation

Implementation. We prototype two versions of SwarmBug to support four swarm algorithms. One in Python (742 lines) to support Adaptive Swarm [2] and another one in Matlab (536 lines) to support Swarmlab [184], Fly-by-logic [132], and Howard's [72]. We also modified existing simulators/emulators. Our analysis for SVMap (Section 2.4.2) is written in R (632 lines).

Environment Setup. We performed our evaluation on an Intel i7-9700k 3.6Ghz and 16GB RAM, and 64-bit Linux Ubuntu 16.04. For the real-world experiment in Section 2.5.2, we use six

Crazyflies [23].

Table 2.1: Selected Algorithms for Evaluation

Name	SLOC	Drones	Objective
Adaptive Swarm [2]	3,091	20	Flight avoiding static & dynamic obst.
Swarmlab [184]	13,213	20	Flight avoiding static obstacle
Fly-by-logic [132]	13,244	6	Optimizing path avoiding unsafe zone
Howard's [72]	1,989	20	Flight avoiding static obstacle

Swarm Algorithms. As shown in Table 2.1, we use four representative and diverse swarm algorithms. To select the four algorithms, we search total 23 swarm-related research papers with open sourced algorithms and 54 public GitHub repositories related to swarm robotics from 2010 to 2020. Among these, 25 came with runnable code from which we pruned out 12 that were just off-line planning algorithms not reactive to the environment, and 9 algorithms that did not exhibit collective behaviors (e.g., collections of individual drones without cooperative interactions). Finally, we end up with the selected four swarm algorithms. Details can be found in [166].

Note that while there are many swarm algorithm papers, the viable implementations are limited. We found that many repositories do not include the full implementations to support the swarm [75, 6, 141, 189, 135, 165] or do not release enough details for usage [210, 8, 149]. Others just include rudimentary implementations that do not provide basic swarm functionality or testing environments (e.g., maintaining formation, avoiding obstacles) [201, 198, 120, 154, 73, 14, 115, 183].

Table 2.1 shows the SLOC (Source Line of Code) of algorithms and the number of drones we used for the evaluation. We use 20 drones for all the algorithms, except for the Fly-by-logic as it does not support a swarm with up to 6 drones. The last column briefly describes the objective of each algorithm. Among the four algorithms, Adaptive Swarm is the only algorithm that enforces a particular formation during the mission. Swarmlab tries to match the speed with other robots during the mission, while the other three algorithms consider other robots as an object to avoid. Swarmlab implements two swarm algorithms: Olfati-Saber's [129] and Vicsek's [181]. We use Olfati-Saber's algorithm because Vicsek's algorithm has a bug (all the robots are disappearing after a mission starts).

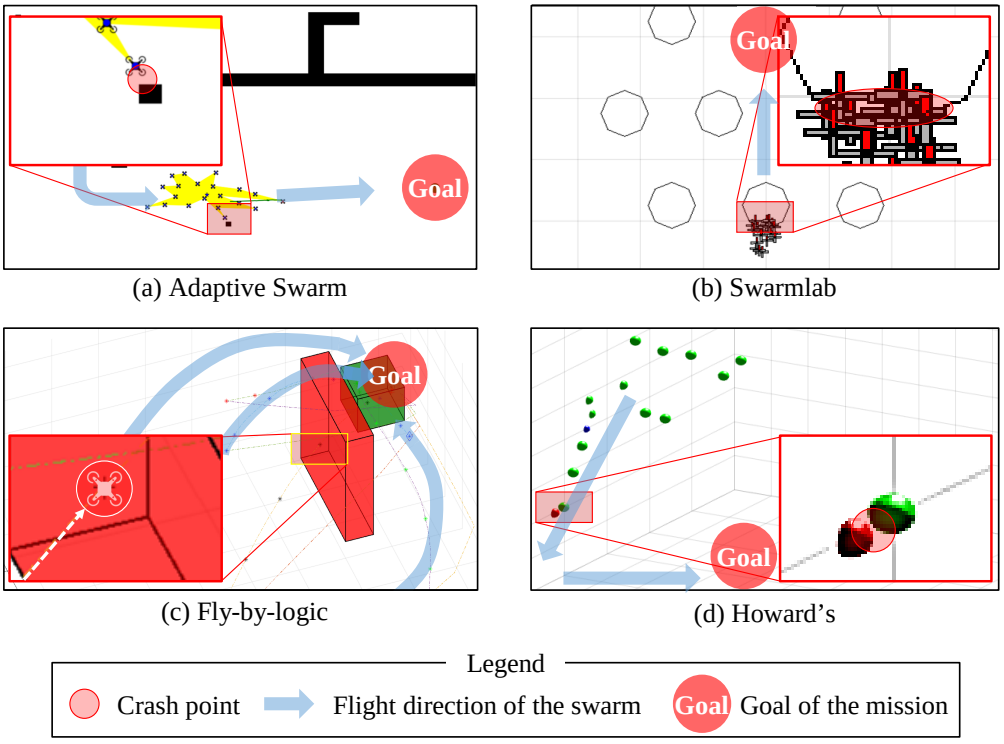
Table 2.2: Effectiveness of SWARMBUG

Algorithm	Trend	Behavior causal analysis: Strategies: Swarm Configuration		Reinforcing	Inverting	Fix validation: Profiling[1] R²	Profiling[1] In²	Fuzzing[3] Reinforcing	Fuzzing[3] Inverting	Fix prioritization: MSE score (Rank) R²	MSE score (Rank) In²	Dev. cfm.[5]
Adaptive Swarm	Decreasing (*robot1.sp*)	*w*	(=20.0)	✔ (+20.0)	✔ (-18.0)	34	16	1195/4292 (28%)	601/4282 (14%)	-	-	-
		xi	(=400.0)	✔ (-380.0)	✔ (+400.0)	100	13	4281/4324 (99%)	477/4333 (11%)	0.024 (2)	-	✔
		nu	(=1.4E-03)	✔ (+1.4E-03)	✔ (-1.12E-03)	100	51	4060/4215 (96%)	1858/4424 (42%)	0.031 (3)	-	✔
		int_dist	(=0.7)	✔ (+0.7)	✔ (-0.56)	88	58	3922/4466 (88%)	2131/4441 (48%)	0.053 (4)	-	✔
		infl_radius	(=0.3)	✔ (-0.24)	✔ (+0.3)	11	100	411/4190 (10%)	4199/4199 (100%)	-	0.022 (1)	✔
		drone_vel	(=4.0)	✖	✔ (+4.0)	-	76	-	3052/4788 (64%)	-	0.061 (5)	✔
Swarmlab	Decreasing (*p_swarm.u_ref*)	*c_vm*	(=3.0)	⊗	✔ (-2.4)	-	15	-	669/4554 (15%)	-	-	-
		b	(=5.0)	✔ (-4.0)	✖	22	-	1019/4323 (24%)	-	-	-	-
		r0	(=10.0)	✔ (+10.0)	✖	100	-	4508/4537 (99%)	-	0.021 (1)	-	✔
		c_pm_obs	(=5.0)	✖	✔ (-4.0)	-	57	-	2311/4661 (50%)	-	-	-
		d_ref	(=10.0)	✖	✔ (-8.0)	-	29	-	1167/4551 (26%)	-	-	-
		v_ref	(=6.0)	✔ (-4.8)	✖	100	-	3811/4088 (93%)	-	0.023 (2)	-	✔
Fly-by-logic	Decreasing (*obs*)	*max_vel*	(=0.8)	✔ (+0.8)	✖	78	-	3776/4896 (77%)	-	0.021 (2)	-	✔
		max_accl	(=1.0)	✔ (+1.0)	✖	60	-	2808/4888 (57%)	-	0.025 (3)	-	✔
		C	(=50.0)	✔ (+50.0)	✔ (-40.0)	100	23	4808/4901 (98%)	-	0.015 (1)	-	✔
Howard's	Decreasing (*wypt*)	*dist_thresh*	(=2.0)	✔ (+2.0)	✖	26	-	1444/6281 (23%)	-	-	-	-
		obst_pot_c[4]	(=1000.0)	✔ (+1000.0)	✔ (-800.0)	100	14	5697/6311 (90%)	831/6211 (13%)	0.011 (1)	-	✔

1: Data in Profiling column indicates the number of successful mission for 100 tests. 2: R and In indicate Reinforcing and Inverting, respectively. 3: Data in Fuzzing column indicates the number of successful mission over the number of fuzz testing in given time and success rate. 4: The program has hardcoded constants instead of variables. We assign a conceptual name to them. 5: Checkbox in this column indicates whether the bugs and fixes are confirmed by developers or not.

2.5.1 Effectiveness

Buggy Behaviors. During the evaluation, we aim to fix four bug classes as shown in Figure 2.9 by using SWARMBUG: (a) A drone fails to avoid a moving obstacle in Adaptive Swarm, leading to a crash, (b) Drones fail to avoid the second static obstacles they encounter, crashing to the pillar structure which is a round shape object in the figure, (c) The first drone fails to avoid the unsafe zone (represented as the red cube) that the algorithm aims to go around, and (d) A drone (the green sphere) crashes into an obstacle (the red sphere).



(a) Adaptive Swarm (b) Swarmlab

(c) Fly-by-logic (d) Howard's

Figure 2.9: Buggy behaviors in the four selected algorithms

Behavior Causal Analysis

Table 2.2 shows the result of SWARMBUG's causal analysis. "Trend" shows the identified trends of Dcc values as described in Section 20. Note that the variable name is the one that dominates the Dcc values. "Strategies" shows all the fixing strategies applied. "Swarm Configuration" shows

the swarm configuration variables (and their initial values) we mutate to apply the fixing strategy (e.g., reinforcing or inverting the trend of Dcc values). To achieve a target Dcc trend, SWARMBUG tries both (1) increasing the value by two times and (2) decreasing the value by 80%, and chooses one that achieves the target Dcc trend. Note that we omit several swarm configuration variables[4] that could not lead to any fixing strategies. Also, some strategies cannot be done by mutating a particular environment variable (e.g., mutating *drone_vel* does not reinforce the trend in the Adaptive Swarm's case). In such a case, we consider the strategy is not applicable and mark it as ✖. Also, there are some cases where the strategies are well achieved while the resulting execution always crashes. To check such a case, we run 10 runs for sanitization purposes. Those that fail to pass the sanitization test (e.g., drones crashing into other objects/drones) are marked as ✪. All successfully applied strategies are annotated by ✔. It does not include the imbalancing strategy which requires the Dcc trends to be balanced, while all the observed Dcc trends are decreasing.

Testing Fixes

"Profiling" presents the results of 100 tests we run for spatial variation profiling (Section 2.4.2). It took approximately 25.2 (for Adaptive Swarm), 2.8 (for Swarmlab), 0.4 (for Fly-by-logic), and 0.3 (for Howard's) hours for run 100 tests. Note that they are naive testing runs where SWARMBUG further conducts fuzz testings (shown in the "Fuzzing" column) guided by MSE scores of Dcc values. In general, our fuzz testing finds more crashes (lower rates of successful runs) than the naive profiling tests, meaning that it is effective in discovering more diverse testing scenarios.

SWARMBUG initially generates 11 (for Adaptive Swarm), 6 (for Swarmlab), 4 (for Fly-by-logic), and 3 (for Howard's) fixes. Gray cells represent fixes that do not fail any tests during the profiling step. "Fuzzing" shows the number of successful tests during the fuzz-testing out of 30 hours for Adaptive Swarm and Swarmlab, 10 hours for Fly-by-logic and Howard's. Gray cells mean the fixes that are most successful (e.g., more than 90% of them are successful). We run Adaptive Swarm and Swarmlab longer than the other two because a single run from the first two algorithms is much

[4] In Table 2.2, we omit 8, 11, 4 swarm configuration variables from Swarmlab, Fly-by-logic, and Howard's respectively.

slower than the other two.

Fix Prioritization

As explained in Section 2.4.3, we obtain MSE scores of the fixes and rank them according to the scores. The most promising fixes are ranked the first in all cases. Two fixes are ranked second: *xi* and *v_ref* in Adaptive Swarm and Swarmlab, respectively. Our manual inspection shows that they are still valid fixes while they are ineffective compared to the fix ranked first.

However, *nu* in Adaptive Swarm, which is ranked third, shows abnormal behavior: it often makes robots stall or even move backward when they recognize obstacles (even if the obstacles are quite far away from them). Our manual inspection reveals that the fix prioritizes avoiding obstacles significantly more than other goals.

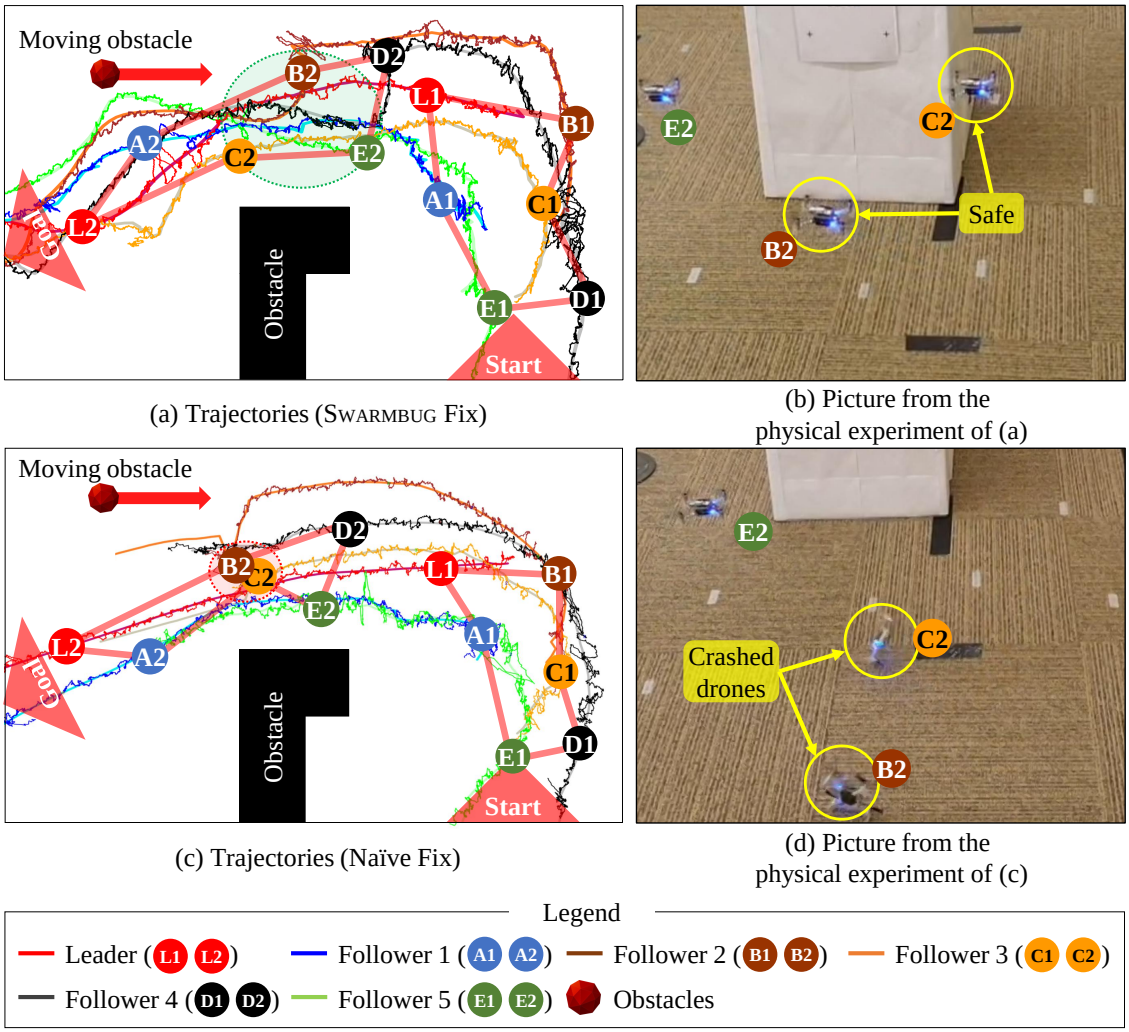
Confirmation from the Algorithm Authors. Throughout our research project, we have communicated with the authors of all four swarm algorithms [2, 184, 132, 72] regarding the configuration bugs we find. The bugs and fixes for the three algorithms are confirmed and acknowledged by the authors. The authors also agreed that the higher-ranked fixes are better than those that are lower-ranked.

2.5.2 Case Study

Real-world Experiment of a Fix from Swarmbug

To show that a fix generated and validated by Swarmbug is effective in real-world environment (e.g., with various noises), we conduct a physical experiment that uses the fixed configuration (*nu*) of Adaptive Swarm to reproduce the same flight.

Setup and Presentation. We use 6 Crazyflies [23] and leverage CrazySwarm [142] as a controller for swarming. We use a local position system (called LPS [24]) supported by Crazyflies to precisely locate drones' 3D positions in space. We conduct the experiments in the lab environment where



(a) Trajectories (SWARMBUG Fix)

(b) Picture from the physical experiment of (a)

(c) Trajectories (Naïve Fix)

(d) Picture from the physical experiment of (c)

Figure 2.10: Trajectories of 6 drones during our physical experiment.

the space is 3m × 4m × 3m (in width × length × height). We use the same trajectory (which includes drones' poses) from the Adaptive Swarm mission shown in Figure 2.9-(a).

Figure 2.10 illustrates the results. Drones start from the right-bottom side of the map (marked as 'Start') and move toward the left (marked as 'Goal'), while avoiding obstacles. There is an L-shape static obstacle which we use two white boxes in our physical experiment. Moving obstacle (i.e., red symbol) is approaching the drones from the left to right direction in the upper side of the map. Thick lines are trajectories computed by swarm algorithms, and thin lines with jitters are the traces of the real physical drones' movements from the motion capture system [24]. The physical

aerodynamics and noise may have caused these variations (i.e., jitters). Along the trajectories, we visualize instances of drones at two different time ticks. Circled letters represent drones, where 'L' means the leader, and A∼E means follower 1∼5. The symbol is followed by a number that represents the time tick of the instances. For instance, 'L1 and A1∼E1' represent the drones' positions at the time tick 1 while 'L2 and A2∼E2' are positions of the same drones at the time tick 2. The red transparent lines between drones visualize a group of drones at the same time tick.

Result. Figure 2.10-(a) shows partial traces of the drones using SWARMBUG's fix "*infl_radius* = 0.6" (from the original value 0.3), which safely finishes the mission without crashing. Figure 2.10-(b) shows a picture of the physical experiment, while safely passing the obstacle (the box behind the drones). With the SWARMBUG's fix, drones maintain a sufficient safe distance. A video of this physical experiment is available on [166].

Finding a Fix without SWARMBUG. To provide a comparison point for the quality of the fix generated by SWARMBUG, we conduct a small additional experiment that tries to come up with a fix by manually changing the parameters without SWARMBUG. First of all, it would take a lot of time to pick the right configuration variable for the fix (i.e., *infl_radius*), without any guidances such as DCC and MSE values used in SWARMBUG. Even if we assume that the desired variable, *infl_radius*, is chosen, finding a good value for the fix is difficult. Assume that 0.4 is chosen (the original value is 0.3). The fix is tested by running the simulations 200 times that are all successfully finished without any crashes.

To this end, we run a physical experiment with the fix as shown in Figure 2.10-(c). Observe that Follower 2 (B2) and Follower 3 (C2) crash each other, meaning that while it passes the naive testing (200 times), the fix is not effective in real-world scenarios.

Debugging a Ground Vehicle Swarm

In this case study, we show how SWARMBUG is used to debug a ground vehicle swarm algorithm's configuration bug. We use a swarm algorithm [179] submitted to an annual robot competition

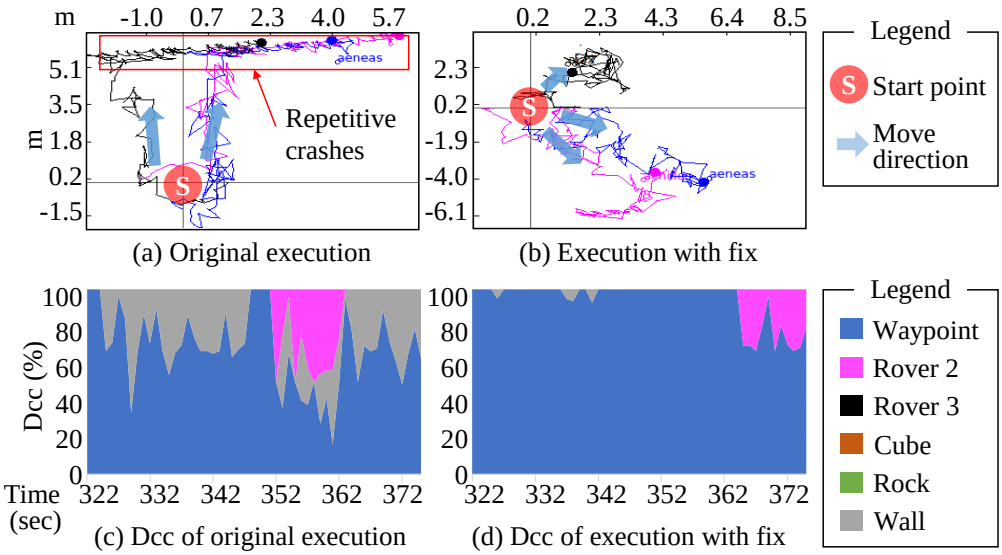


Figure 2.11: Applying SWARMBUG to Swarmathon

funded by NASA: Swarmathon [127, 169]. The algorithm [179] took third place in the competition and was selected as the authors identified the bug with a few test runs. The goal of the algorithm is to leverage the swarm robots to gather resources spread throughout the map quickly.

During the mission, there is a buggy behavior that rovers keep crashing on the north border of the map and get stuck into the north-east corner, as shown in Figure 2.11-(a).

SWARMBUG identifies the Dcc trends as shown in Figure 2.11-(c) with seven environment configuration variables. Fluctuating Dcc values for the wall (i.e., the gray area) in the graph represent the crashes. SWARMBUG applies the *balancing* strategy based on the trend, identifying 14 potential fixes (from 38 swarm configuration variables). Among these fixes, "*M_PI_2 = rand()+pi()/2*" ranked the first (the original value for the variable is "*pi()/2*"). The fix yields the Dcc trend shown in Figure 2.11-(d). Most of the gray area is removed as the fix reduces the number of crashes. As shown in Figure 2.11-(b), the execution with the fix does not show the buggy behavior (e.g., drones stuck in the corner).

2.6 Discussion

Overhead. During the operation, SWARMBUG runs a number of tests and conducts various analyses (e.g., computing DCC and MSE values) on the collected data from the tests. Note that the analyses are done offline. We also instrument existing simulators to collect values for DCC computation and the instrumentations incur less than 5% overhead at runtime.

Applicability of SWARMBUG's Fuzz Testing. While this research focuses on finding and fixing configuration bugs, SWARMBUG's fuzz testing can find other types of bugs as well.

Specifically, while fuzz-testing the Adaptive Swarm, we find a bug in the algorithm that may rarely appear at runtime. That is, when a follower drone and the leader drone get very close to each other, the leader does not try to avoid the follower, leading to a crash. Our manual analysis shows that the leader drone's algorithm does *not consider follower drones as an object to avoid*. This is odd because follower drones have the logic to avoid the leader drone if they get too close. Our conversation with the developer confirmed that the developer assumed that the leader will always be far ahead of other drones and do not need to implement code to avoid a collision. Even testing three days without SWARMBUG does not reveal the bug. SWARMBUG's fuzz-testing identified such a scenario and exposed the defect, thanks to the guidance via DCC and MSE values. We also validated this can happen in the real-world and the issue is confirmed by the author of the algorithm as well. More details can be found on our project page [166].

We also find that DCC can be used to identify buggy logic in the swarm algorithm. Specifically, when we initially evaluate Howard's algorithm, we find that SWARMBUG could not find any possible fix. We investigate the DCC values produced during the experiment further and notice that the observed DCC values are extremely stable, except for slight variations observed in *obst_pot_c* just before the drone crashes. As we trace back to code related to *obst_pot_c*, we found that it detects the obstacle only after a crash happens. To properly avoid objects before it crashes, the algorithm should detect the object before it gets too close.

To fix this, we modify the algorithm so that it can detect objects early. After we patch the algorithm

(can be found on our project page [166]), we conduct our evaluation on the algorithm again, and SWARMBUG successfully finds a possible fix as shown in Table 2.2.

Scalability and Usability of SWARMBUG. Our design is general and applicable to other swarm algorithms while it requires some engineering effort. Specifically, to support a new swarm algorithm, two tasks are required: (1) identifying configuration-variables (we provide a profiling tool for this in Section 2.4.1) and 5 thresholds (e.g., mission completion time, time-window, MSE thresholds), (2) instrumenting the algorithm to integrate SWARMBUG (e.g., changing 289 SLOC for Adaptive Swarm). In our evaluation, it took 10∼18 hours (by a graduate student with moderate experience in drones) to complete the two tasks for an algorithm. The effort is non-trivial, but it is required one time for each algorithm. For example, besides the four evaluated algorithms, we have applied SWARMBUG to Swarmathon (see Section 2.5.2), taking about 10 hours (identifying 38 configuration-variables, the 5 thresholds, and changing 152 SLOC for the integration).

Future Directions. We envision future directions of our research along two dimensions: empirical and technical. For the empirical aspect, applying SWARMBUG to more diverse swarm algorithms/systems (e.g., ground vehicle swarm) and more complicated scenarios (e.g., drones navigating a city landscape) and analyzing the cost and benefits of it can be the future work. Also, further analysis support to complete some of the semi-automated processes such as identifying key parameters used as inputs in SWARMBUG can be the future work as the technical aspect.

2.7 Related Work

Testing Autonomous Robotics. Several testing methods are proposed [70, 9, 76] and studied [1] to solve and understand diverse challenges in testing autonomous robots. To evaluate the exploration of the system under test (SUT), coverage-driven verification (CDV) guides the testing process with an automated and systematic aspect; thus developers generate a broad range of test cases [9]. ASTAA [76] proposed an automated system specialized in stress and robustness testing and then discovered hundreds of bugs. Timperley *et al.* empirically studied and found that the majority of bugs in autonomous systems can be reproduced by software-based simulations [174].

Hildebrandt *et al.* integrated dynamic physical models of the robot to generate physically valid yet stressful test cases [70].

Alternatively, formal validation and verification are rigorously studied [79, 114] and used to prove properties of the testing programs such as correctness, functionality, and availability. Bensalem *et al.* developed a toolchain for specifying and formally modeling the functional level of robots [21], and Halder *et al.* implemented a system for checking the model of robots. Deeproad [47] validated inputs for testing autonomous driving systems.

Unlike previous studies, SWARMBUG aims to debug swarm algorithms, which is an order of magnitude more complex, by using the novel concept of the degree causal of contribution (DCC).

Testing/Debugging Approaches.

Delta debugging [203] isolates the difference between a passing and a failing test case, by running mutated test cases and observing the execution results. BugEx [148] and Holmes [83] leverage a similar approach to understand the cause of bugs.

In addition, Coz [41] introduces additional delays to infer possible optimization opportunities. LDX [100] perturbs program states at runtime to infer causality between system calls.

SWARMBUG uses a similar idea of mutating environment configuration variables to conduct behavior causal analysis. However, SWARMBUG handles swarm algorithms where inputs are essentially streams of data, while other techniques may need a non-trivial amount of modifications to handle such input data. SWARMBUG also leverages the DCC values to create a fuzz testing system.

Researchers leveraged random testing techniques (e.g., fuzzing) to continually improve the quality of test cases [140, 102, 175]. PySE [96] used a reinforcement learning-based approach to find a worst-case scenario. There are also model-based approaches inferring the actual program state [153, 160] or input types [186].

Automated Program Repair. There is a line of research focused on fixing buggy programs automatically [62, 103, 64, 187, 88]. In particular, [103] leverages a genetic programming approach [97]

to repair a buggy program. [64, 187] proposes an automated program repair technique for programming assignments. While the previous works and SWARMBUG share the same goal of fixing a bug, SWARMBUG aims to fix configuration bugs in complex swarm algorithms running multiple robots. It fixes bugs by changing the swarm configuration variables' values, while the previous works change the program code to repair. QLOSE [42] leverages program distances to come up with solutions for program repairing. SemCluster [138] defines a new metric based on the input data space and uses the metric to cluster programs. SWARMBUG leverages DCC to guide the analysis and testing for swarm algorithms.

2.8 Summary

We proposed SWARMBUG, a debugging approach for resolving configuration bugs in swarm algorithms. SWARMBUG automatically identifies the causes of configuration bugs by creating new executions with mutated environment configuration variables. It compares the new executions with the original execution to find the causes of the bug. Then, given the cause, SWARMBUG applies four different strategies to fix the bug by mutating swarm configuration variables, resulting in fixes for the configuration bugs. Our evaluation shows that SWARMBUG is highly effective in finding fixes for diverse configuration bugs in swarm algorithms.

Chapter 3

SwarmFlawFinder: Discovering and Exploiting Logic Flaws

3.1 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 research 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 [116, 190, 188], 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 [2, 68, 39, 33], 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 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 [170].

3.2 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 research, 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 [61, 112, 202] 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 [59, 15] often require expensive analyses (e.g., symbolic analysis) on the swarm algorithm. Blackbox approaches [19] 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 [195, 152, 69, 172, 65], where most of them focus on the modeling of swarms, and their discussions are at a high level. Specifically, Winfield et al. [195] 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 [66]. They also discuss difficulty in testing such as the large input space. Higgins et al. [69] 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 [152] present models of malicious swarms aiming to make a victim swarm operation inefficient.

Compared to the above work [195, 69, 152], we aim to identify *concrete logical flaws from real algorithms* via testing. In the context of [195], 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 [195]. Note that while [69] 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 [69]'s definitions (In Table 3.3, 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, [152] 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 3.3). However, [152]'s discussions are conceptual and all the discovered flaws we find are new. Note that the models in [195, 69, 152] 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. [172] 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 this research, 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. [65] 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 [87, 156] and network packet injections are not our focus.

3.3 Motivating Example

We use a drone swarm mission running Adaptive Swarm [2] 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 Figure 3.1-(a). Each drone is attached with a string to hold the object. Typically, it takes 189.4 ($\pm$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.

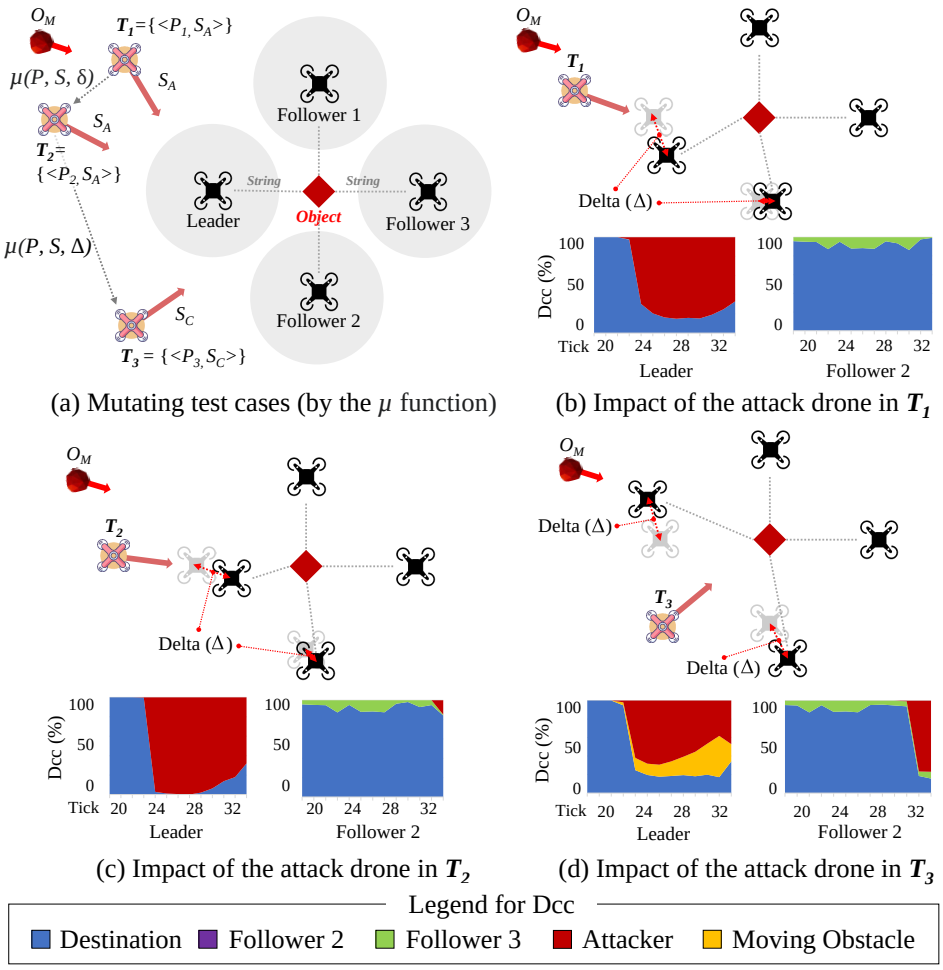
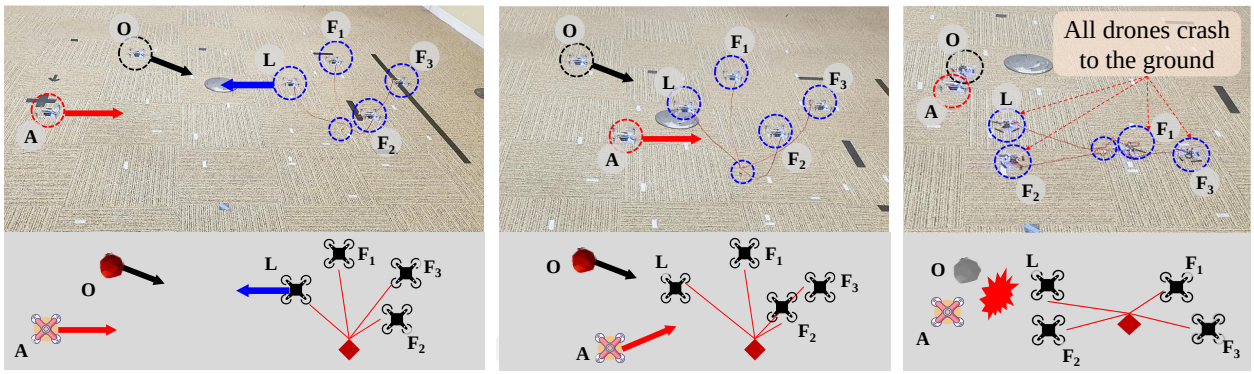


Figure 3.1: SwarmFlawFinder in action on the motivation example.

1) Test Creation: Given the target swarm mission, we create the initial test (T_1 in Figure 3.1-(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 Figure 3.1-(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 direction and S_3 represents a strategy that moves between two victim drones. Other strategies can be found in Section 3.4.1.



(a) Attack drone and obstacle approach the victim swarm

(b) Attack drone influences a victim drone

(c) Leader drone crashed into the obstacle

Figure 3.2: Physical experiment reproducing the crash shown in Figure 3.3 (L means Leader and $F_{1\sim3}$ indicates Follower 1∼3).

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 [104, 100], 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 Section 3.4.2). 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 Figure 3.1) are collected. We repeat this for all external objects, and accumulate the Δ values to get the Dcc values, shown at the bottom of Figure 3.1-(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

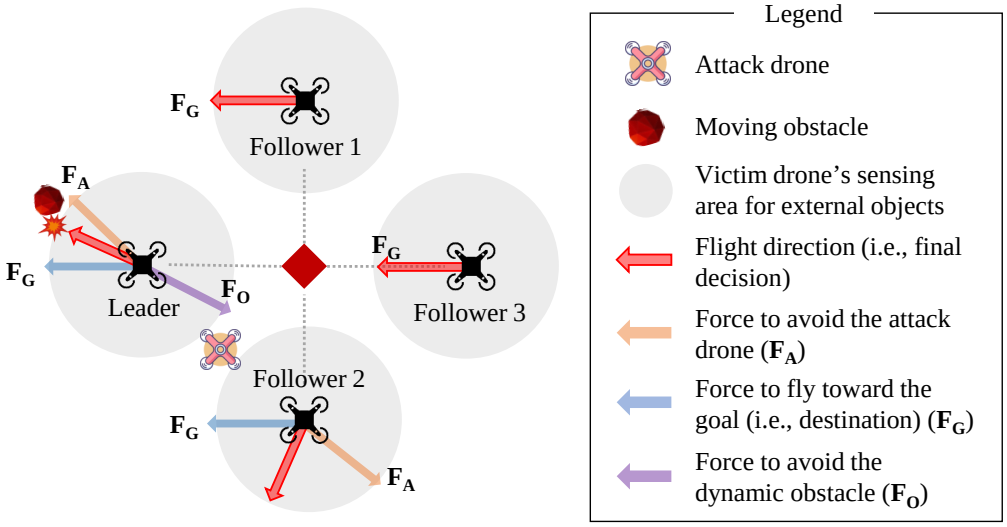


Figure 3.3: Crash (caused by a logic flaw) found by SwarmFlawFinder.

current test. It derives the next test by mutating the test case slightly, denoted by $\mu(P, S, \delta)$. Observe that T_1 and T_2 in Figure 3.1 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 Figure 3.1-(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. Figure 3.3 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

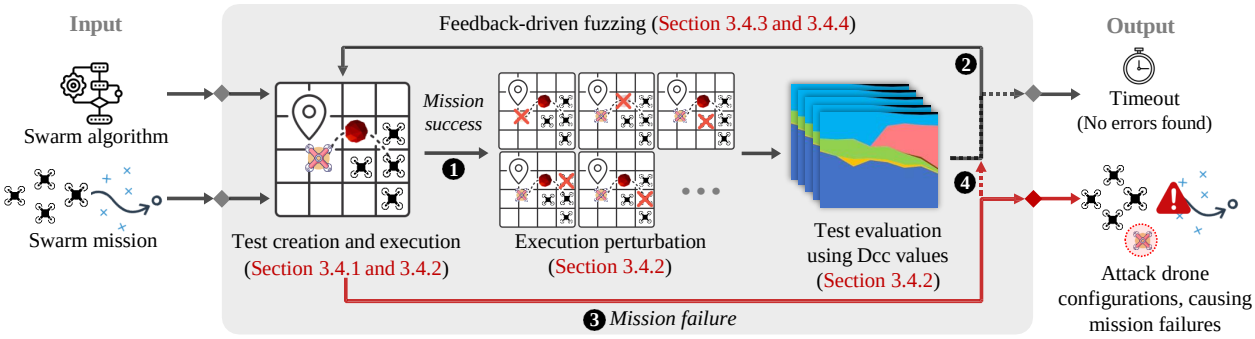


Figure 3.4: Overview of SwarmFlawFinder. (The shaded area represents SwarmFlawFinder with input and output on the left and right, respectively)

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 Figure 3.3), 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 Figure 3.2. 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). Figure 3.2 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 [2] 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 Section 3.5.2 (*C1-2. Naive multi-force handling*).

3.4 Design

Figure 3.4 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 (Section 3.4.1). If a test mission finishes successfully (❶), it conducts execution perturbation (Section 3.4.2) 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, Section 3.4.3). 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.

3.4.1 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 Section 3.4.4.

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 $\times$ width $\times$ 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 [170] or Section A.4).

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.

3.4.2 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 [104, 100] which explains the meaning of causal claims in terms of counterfactual conditionals: "If X had not occurred, Y would not have occurred."

> **Counterfactual Causality** [104] 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}.

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.

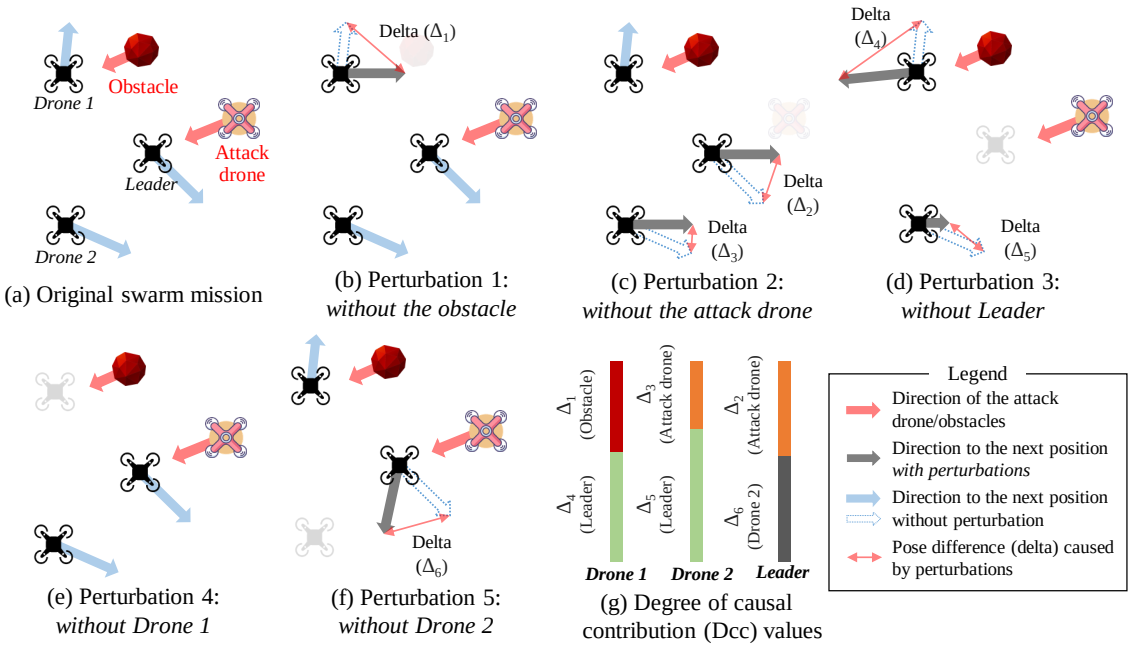


Figure 3.5: Dcc computation via perturbed swarm executions.

SwarmFlawFinder perturbs all objects including victim drones, objects, and attack drones, one by one in each perturbed execution. Figure 3.5-(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.

Figure 3.5-(g) shows Dcc values computed from the perturbed executions at the moment illustrated in Figure 3.5. 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 2 shows the algorithm to compute Dcc values. Specifically, 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 Figure 3.5). 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).

3.4.3 Dcc Guided Fuzz Testing

Abstracting Swarm Missions via Dcc. Figure 3.6 shows a series of Dcc values computed throughout the swarm mission (0∼180 ticks). X-axis and Y-axis represent the time and stacked

Algorithm 2 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
```

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) [105] which is commonly used to compute the similarity between data in various fields [199, 177, 146]. Figure 3.6 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.

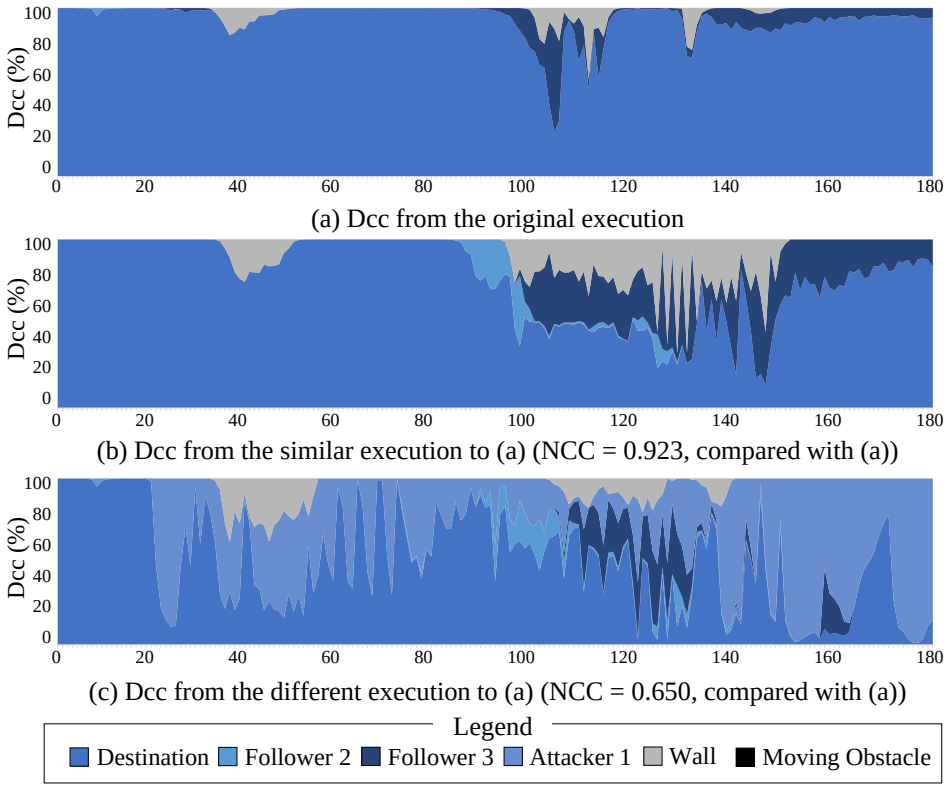


Figure 3.6: Example of NCC scores from three executions.

Using NCC [105] 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 2). 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 research, line 22 in Algorithm 2), 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 3.2.

Algorithm. `FuzzTesting()` in Algorithm 2 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).

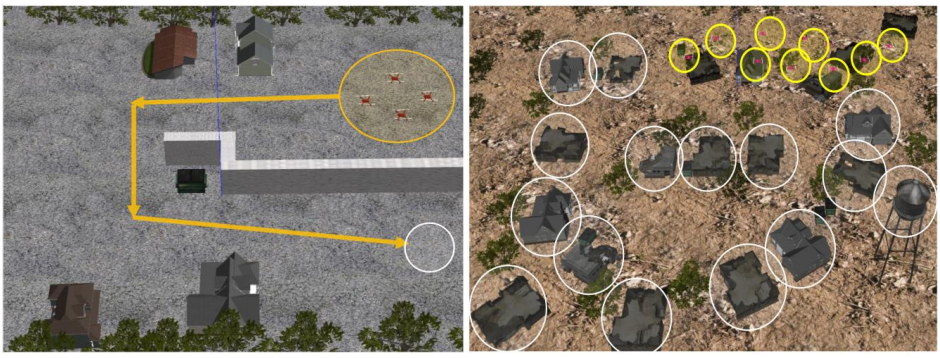
3.4.4 Testing with Multiple Attack Drones

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 can be found in [170] or Section A.9.

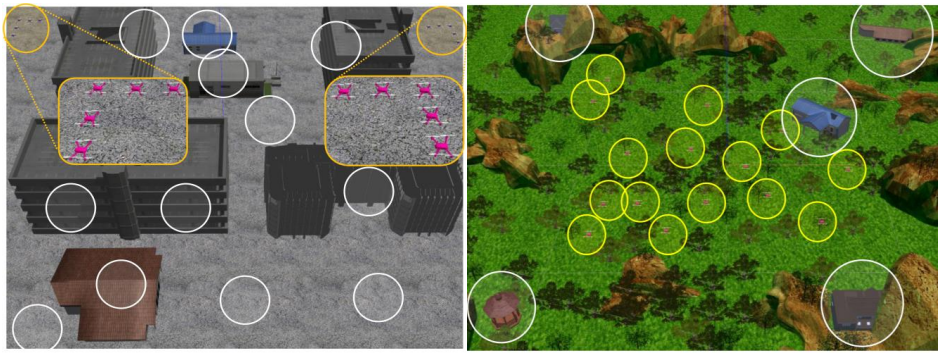
Mutating Tests with Multiple Attack Drones. As described in Section 3.4.1, a test run with multiple drones is defined as a test case with multiple tuples such as $\{< P_1, S_1 >, < P_2, S_2 > , ..., < P_n, S_n >\}$, where each tuple represents an attack drone. When there are multiple attack drones in 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 2). 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.

An example scenario with multiple attack drones can be found on [170] or Section A.5.



(a) Adaptive Swarm (Navigation) (b) SocraticSwarm (Coordinated search)



(c) Sciadro (Distributed search) (d) Pietro's (Search and rescue)

Figure 3.7: Visualizations of the selected algorithms' missions. Yellow and white circles indicate swarm drones and search/rescue targets or the destination.

3.5 Evaluation

3.5.1 Experiment Setup

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 [72, 183, 73, 14, 115, 197, 67, 136, 22, 55] are not considered.[1] To this end, we choose four runnable algorithms that exhibit collective swarm behaviors and allow us to introduce external objects.

Selection Criteria

As shown in Figure 3.8-❶, 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).

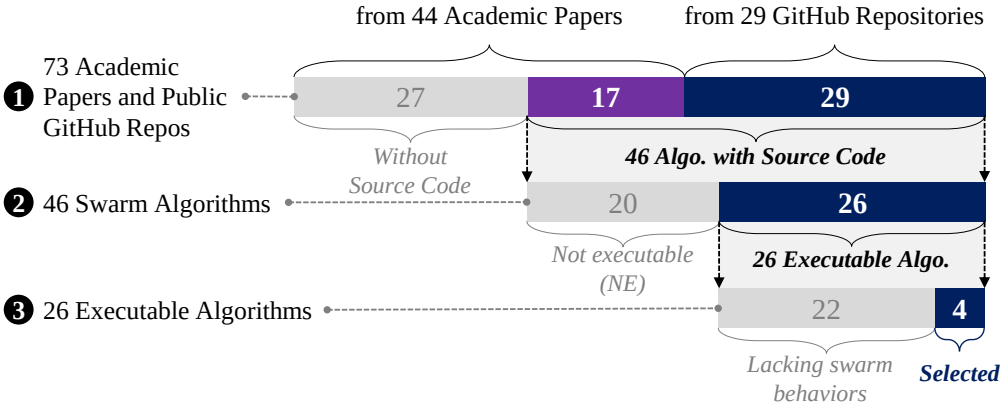


Figure 3.8: Algorithm Selection Process

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 [170].

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

[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.

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 [170].

Sizes of the Algorithms. Figure 3.9 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.

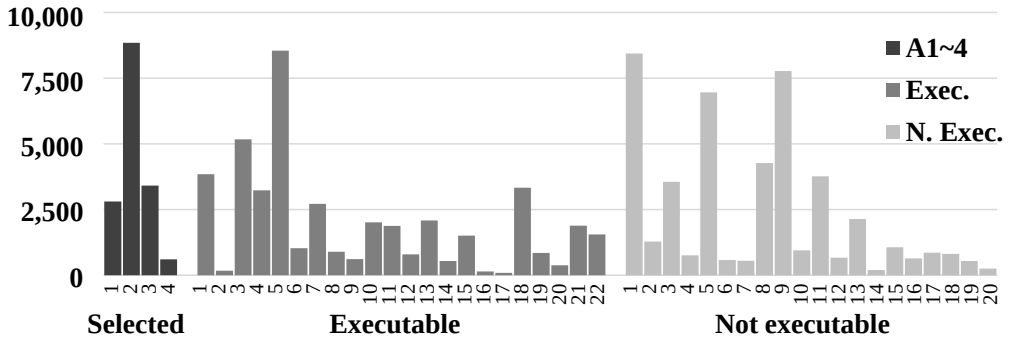


Figure 3.9: SLOC of Considered and Selected Algorithms. Avg. of A1-4: 3,919 lines, Executable: 1,968 lines, and Not Executable: 2,305 lines.

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 [44] conducted swarm missions aligned with our selected swarm algorithms: searching missions in urban/rural areas [45]. 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 [54] 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 [119] releases its source code [158]. 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 [33] might be used in proprietary products, while the authors choose not to reveal the details.

Table 3.1: Selected Swarm Algorithms for Evaluation

ID	Name	SLOC	Language	Algorithm's Objective
A1	Adaptive Swarm [2]	3,091	Python	Multi-agent navigation
A2	SocraticSwarm [68]	9,920	C#	Coordinated search
A3	Sciadro [39]	3,851	Netlogo	Distributed target search
A4	Pietro's [33]	752	Matlab	Coordinated search and rescue

Selected Target Algorithms. Table 3.1 presents the selected four swarm algorithms and Figure 3.7 shows visualizations of the swarm algorithms using the Gazebo simulator [3].

A1. Adaptive Swarm [2] aims to move a swarm of (up to 20) drones, from the current position to a predefined destination (shown as a yellow path in Figure 3.7-(a)) while maintaining a formation and avoiding obstacles.

A2. SocraticSwarm [68] conducts a swarm searching mission, where individual drones actively interact with neighbor drones to share information, as shown in Figure 3.7-(b).

A3. Sciadro [39] runs multiple swarms to search targets distributed over a wide range of areas, as shown in Figure 3.7-(c). Swarm groups can be dynamically changing at runtime, allowing individual drones joining and leaving a swarm.

A4. Pietro's algorithm [33] aims to achieve a cooperative rescue mission. Figure 3.7-(d) shows an example mission: searching and rescuing targets inside various structures. The process is accelerated with more participating drones.

Table 3.2: 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

Experimental Configurations

Table 3.2 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.

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

Table 3.3: 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**	

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).

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).

3.5.2 Effectiveness in Finding Logic Flaws

Table 3.3 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

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

scenarios. We also communicate with the developers to confirm the fixes. Fixes with '*(Confirmed)*' are the ones that are confirmed.

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 Section 3.5.5.

> *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 involved, as shown in Section 3.3's Figure 3.3.

> *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.

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).

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 Section 3.5.5.

> *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 Section 3.5.5.

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

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 [2, 39]):* We change `repulsive_coef`, `sensing_radius`, and `IR_dist` configuration variables with the values of 400, 10, and 4 respectively. The developers of [2, 39] agreed with our analysis and the fix.

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.

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.

Invidivual Fixes for A1. Table 3.6 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 3.3). 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 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.

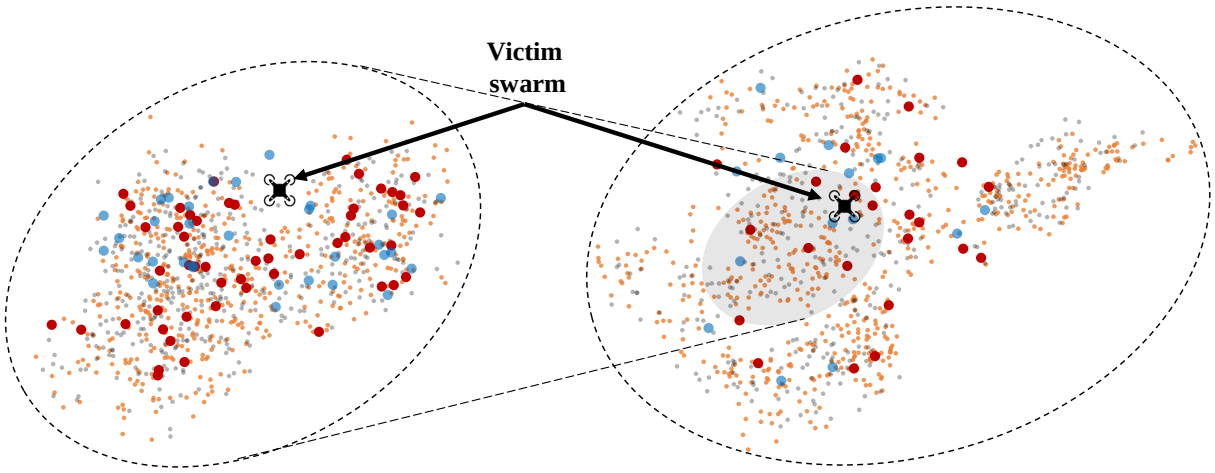
Tuning the Integrated Fix for A1. To make the integrated fix work, we tuned the fix. Specifically, when we combine the individual fixes, we tune the fix for C1-2 and C1-6. The original fixes for C1-2 and C1-6 add `0.15` and `200` to `influence_radius` and `repulsive_coef`, respectively. We reduce the 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 [170] or Section A.6.

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 [170] or Section A.6. 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 a random testing approach

Figure 3.10: Spatial Distribution of Test cases generated by (a) SwarmFlawFinder and (b) the random testing approach.

Influence of Moving Obstacles to Our Evaluation. In our evaluation (Section 3.5), 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 3.4 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).

3.5.3 Effectiveness of Dcc in Fuzz Testing

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

Table 3.4: 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

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 Section 3.4.3.

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. Figure 3.10 shows the results of A1 (Results for A2, A3, and A4 are presented in [170] or Section A.7). Specifically, Figure 3.10-(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 Figure 3.10-(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 Figure 3.10), when we run both for the same period (i.e., 24 hours). We present details of the statistics in the Appendix (Figure 3.11). 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 subset of SwarmFlawFinder's tests.

Effectiveness in Finding Mission Failures

Finding mission failures during testing is critical since they can lead to logic flaws of the algorithms. Figure 3.11 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.

Impact of Searching Space on Random Testing

Observe that the random testing approach's test cases are spread over the wide area in Figure 3.10. 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 shade

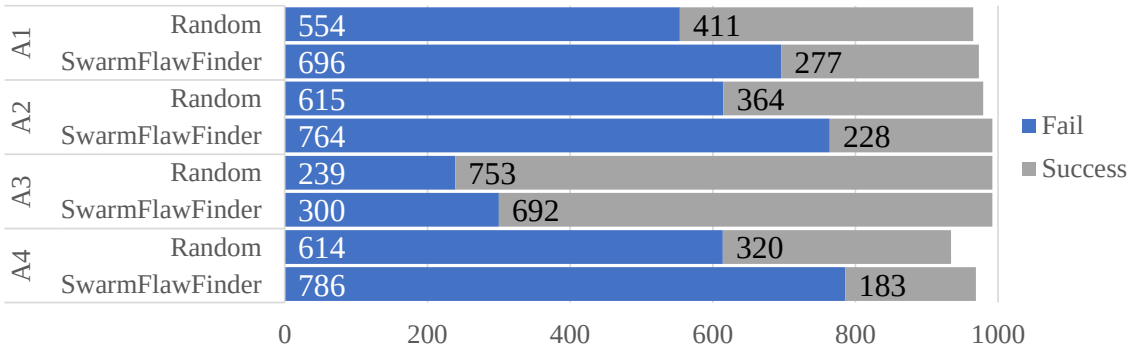


Figure 3.11: Effective test cases (i.e., failures) from the random testing approach and SwarmFlawFinder

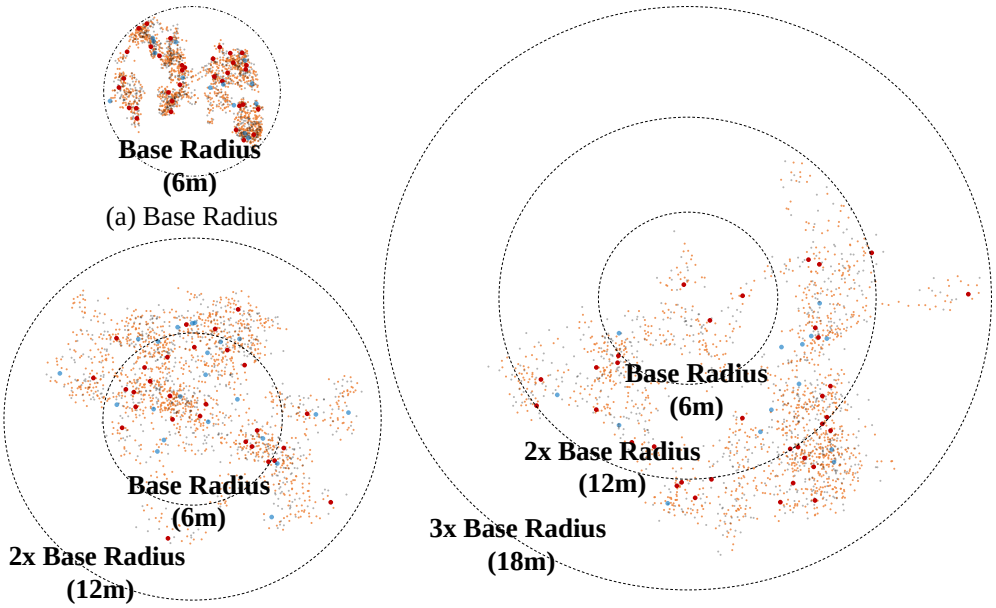
area in Figure 3.10-(b)) obtained by SwarmFlawFinder. We also run two more experiments with 2x and 3x Base searching spaces (as shown in Figure 3.12). The results show that the random testing performs better when given the searching space. However, it still misses three logic flaws C1-2, C1-3, and C2-1, that are dependent on the subtle timing.

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 Section 3.5.1. We define three different searching spaces for each algorithm. First, we run SwarmFlawFinder and obtain the explored space by SwarmFlawFinder as shown in Figure 3.12-(a), considering it the baseline space. Second, from the baseline space, we define 2x and 3x Base (Figure 3.12-(b) and (c)) by extending the radius of the baseline by 2 and 3 times.

Results. Table 3.5 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

Table 3.5: 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



(a) Base Radius

(b) 2x Base Radius

(c) 3x Base Radius

Figure 3.12: Examples of Searching Space Definition from A1. Dots in this figure represent executed test cases with the searching space restrictions.

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.

3.5.4 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 Figure 3.13, SwarmFlawFinder covers two times more Dcc values (avg. 63.5%) than the random testing version (avg. 28.5%).

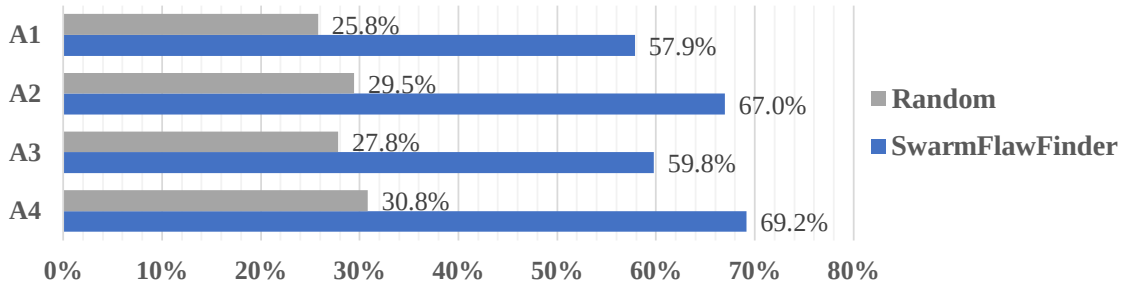


Figure 3.13: Coverage of Unique Dcc Values.

Observed Unique Dcc Values

Figure 3.14 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.

Table 3.6: 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.

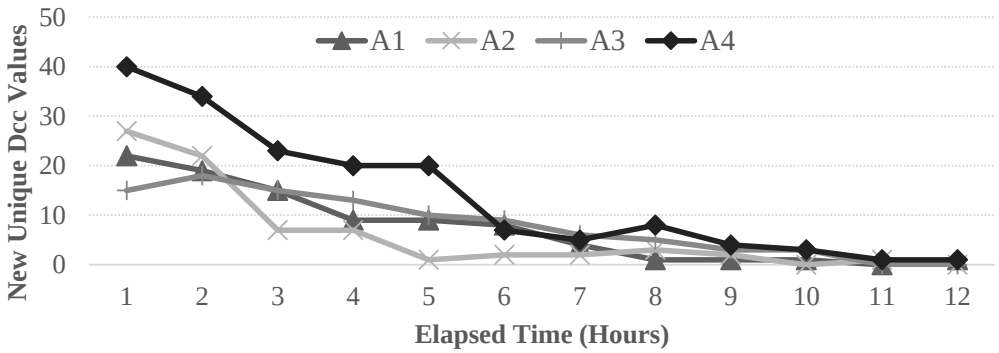


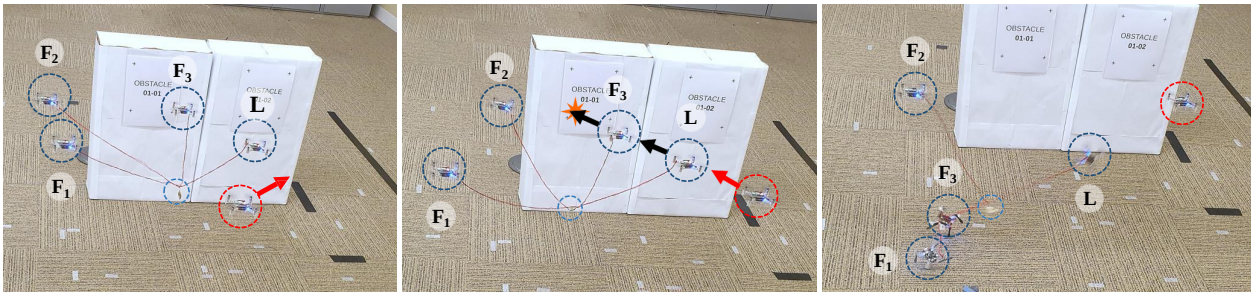
Figure 3.14: Observed unique Dcc values during testing over time

3.5.5 Case Studies

Missing Collision Detection in Adaptive Swarm

Figure 3.15 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 3.3. In Figure 3.15-(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.

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.



(a) Attack drone pushes the leader drone

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

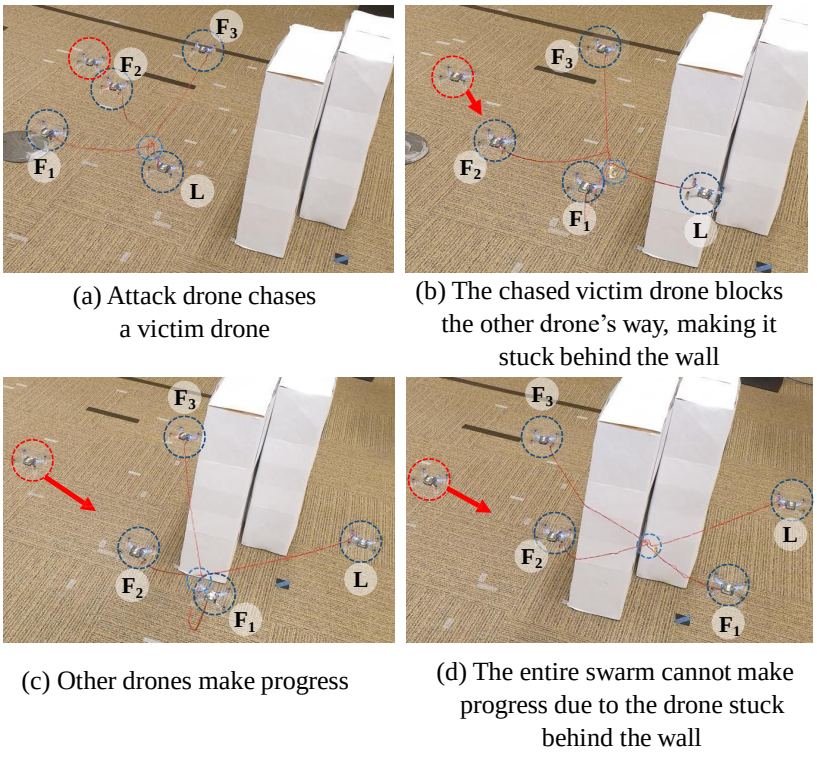
(c) Mission failed with a crash (The attack drone still alive)

Figure 3.15: Attack drone causing a victim drone (F_3) to crash into the wall.

Suspended Swarm Mission due to a Logic Flaw

We find another logic flaw (C1-5 in Table 3.3) in A1. Figure 3.16 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.



(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

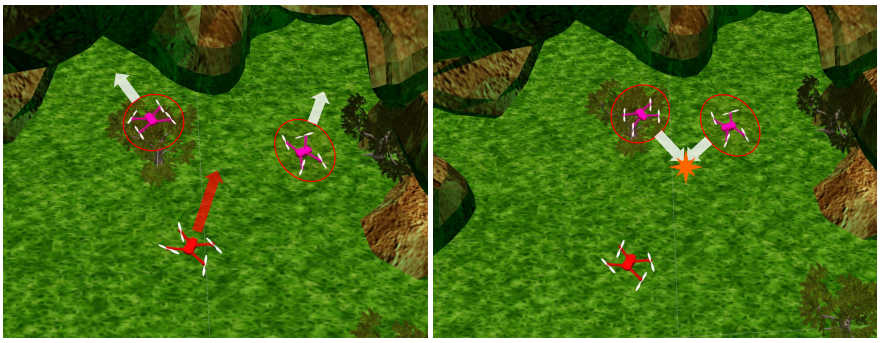
(d) The entire swarm cannot make progress due to the drone stuck behind the wall

Figure 3.16: Attack drone pushes a victim drone F_2 to suspend the swarm's progress.

Detouring without Sensing

Figure 3.17 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.



(a) Victim drones approaching the corner

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

Figure 3.17: Drones crashing while detouring due to obstacles.

3.6 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.

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 [170].

3.7 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 [137, 107, 70, 3, 159, 9, 76, 1], and shown to be effective [174]. [9] 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. [128, 99] apply combinatorial interaction testing to detect flaws triggered by interactions of parameters, while they also require definitions of systems' configuration space. Calò [30] proposes using search-based approach to generate collision inducing configurations for autonomous driving systems. [70] 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. [192] aims to find faults in a flocking algorithm of on ground vehicle swarms by using genetic algorithms (GA) [50]. 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 this research 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 [21, 47, 32, 207, 31, 48]. 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 [61, 112, 202] fuzzers, while others [116, 131, 34, 53, 89] 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 [116, 200, 35] 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 [90, 150, 93, 7, 49, 71]. 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 approach. Moreover, existing fuzzers [90, 150, 93, 7, 49, 71] 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. [7, 49, 71] aim to detect general type bugs only (e.g., buffer overflow). [93] can only detect limited types of misbehavior (e.g., finding input validation bugs). [90] relies on substantial domain knowledge, which is not designed for swarm robotics. Others [150, 49, 71] also focus on bugs related to a specific environment, such as weak ports [150], MAVLink protocol [49], and WiFi [71]. 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 [180, 162, 143] and defenses [36, 145, 125, 122, 18, 37, 133] of drones have gained significant attention. There are testing tools [123] 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.

3.8 Summary

This research 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.

Chapter 4

SwarmGen: Generating Challenging Environments for Swarm Testing

4.1 Introduction

Swarm robotics is an emerging research field that studies how multiple robots can be used to solve collective tasks, which are challenging for individual robots [151, 26, 20], such as environmental monitoring, search, and rescue. Under the hood, a *swarm algorithm* is the core decision-making component that controls and coordinates multiple drones. Testing a swarm algorithm is crucial for developing robust drone swarms [84, 1, 9]. Considering drones are highly reactive to various environmental factors [26, 126] and swarm algorithms make extremely dynamic decisions based on them [17, 86], the environment to interact is crucial part of swarm algorithm testing. However, the problem is most of the provided default environment is simple and focuses on the basic features of the swarm, which is not complex enough to cause various behaviors. To this end, the developer may want to generate a more complex environment for swarm testing. However, it is challenging due to the following two major reasons. First, generating a complex environment requires enough domain knowledge about the interaction of the target swarm algorithm and the environment. Second, even if one who wants to generate the mission environment has enough domain knowledge, trial-and-error is required to reach the complex environment. Hence, generating a complex mission environment is challenging especially when a swarm operation is sophisticated [1, 38, 28].

In practice, generating a complex swarm mission without any guidance is costly, meaning that a

substantial amount of time and resources might be wasted in an undesirable environment. For example, given a default mission, a developer may want to change or add obstacles in the mission environment to induce more various behaviors, including buggy behaviors. Even if the developer observes the swarm's behaviors, it is difficult to know the impact of the changed environment on the behavior. To this end, the ability to measure the interaction between the swarm and the environment is required to know whether an introduced change makes the environment more complex or not. However, measuring the interaction is challenging as there are many factors that impact the swarm's behavior at the same time.

This research explores a systematic approach for mission environment generating for swarm algorithm testing. Our key idea is that a more complex environment causes more complex behavior (e.g., unexpected behavior leading to bugs). If we observe a more complex swarm behavior from a current environment, we consider it a more complex environment. To this end, we leverage a metric called the degree of causal contribution (or Dcc) used in [84] to capture swarm behaviors in the environment. It captures the fine-grained causal relationships between external factors and the swarm flight by running experiments without each potentially contributing factor. Leveraging Dcc, we identify more complex swarm behaviors and further compare the environments.

We evaluate our approach using 10 missions on 4 swarm algorithms [2, 164, 117, 209]. The result shows that our approach generates complex environments for swarm testing successfully. Moreover, compared to testing using random generation, testing using our approach discovers 44 more unique behaviors, including 13 more bugs than using the original mission environment.

Our contributions are summarized as follows:

- To the best of our knowledge, we develop *the first automated swarm mission environment generation* technique based on analysis of swarm behaviors.
- We leverage the degree of causal contribution (Dcc) to abstract the swarm behaviors as the reflection of the environment complexity on a swarm operation.
- We evaluate our approach on 4 swarm algorithms with 10 missions, and our testing using

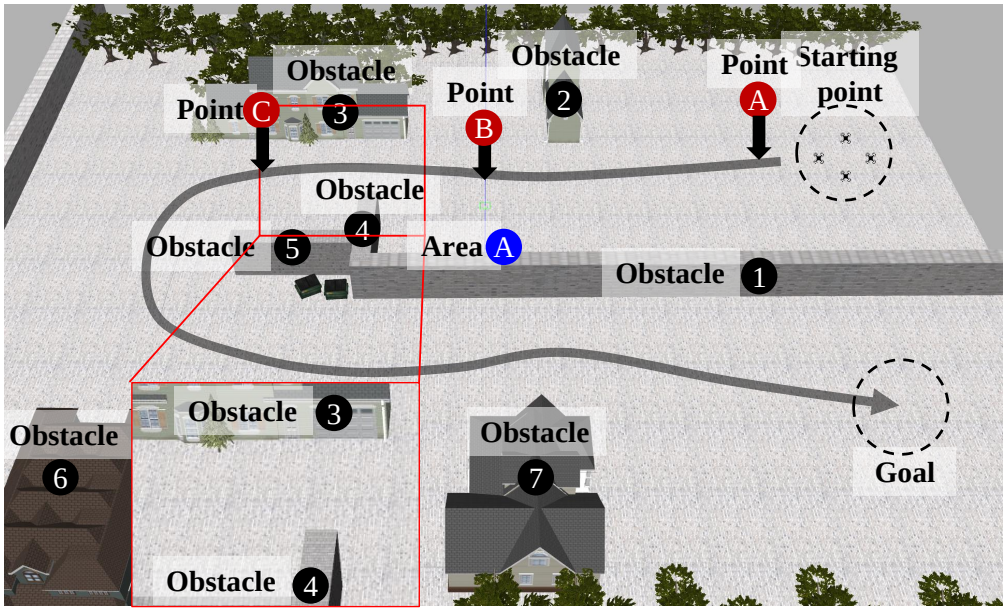


Figure 4.1: Motivating example mission.

generated complex environments discovers 44 more unique swarm behaviors including 13 more bugs compared to testing with the default environment.

- We publicly release the source code of our prototype and data on [171].

4.2 Motivating Example

We use a drone swarm mission running Adaptive Swarm [2] to illustrate our approach in action. Figure 4.1 shows the target swarm mission that aims to reach the goal from the starting point while avoiding obstacles and maintaining the formation of the swarm. Assume that a developer wants to test the swarm algorithm and find buggy behaviors to fix them. The given default mission environment is enough to demonstrate the basic features of the swarm, such as navigating and avoiding obstacles, but it is too simple to observe various swarm behaviors. For this reason, the developer tries to make a more complex environment by modifying the current environment, maintaining the original goal of the mission (i.e., removing the obstacle ❶ goes against the original testing purpose). A naive way to achieve it is to manually change (e.g., add, remove, or resize)

the obstacles on the route of a swarm in a mission. However, the swarm does not work properly in the changed environment (e.g., blocked by newly introduced or modified obstacles) or does not show the differences. The developer may perform many trial-and-errors, but it consumes a lot of engineering effort even if the developer has enough domain knowledge. Moreover, it is difficult for the developer to know whether the currently changed version of the environment is the most complex or not. Hence, we aim to obtain a complex mission environment automatically that induces more diverse interactions between the swarm and the environment to observe corner cases effectively. To this end, it is critical to *measure the swarm behaviors*.

Challenges. A swarm's behavior is an outcome of various factors; hence, capturing its behavior requires *comprehensively considering all the factors*. While there are various metrics for measuring the swarm behavior [5, 178, 185, 25, 82, 111], they are not sufficiently comprehensive. For instance, we present the following three types as examples:

1. *Accuracy:* [5, 178] propose a metric called *accuracy*. It is computed from differences between each drone's actual direction (i.e., angle of the flight) and planned direction (i.e., the value is lower if the differences are large). We obtain the accuracy value for a swarm by taking an average of each drone's accuracy.

2. *Coherence:* [185, 25] introduce the *coherence* metric that represents the differences between the drones' flight directions. The value range is 0 to 1, where if all drones fly in the same direction, the value is 1.

3. *Swarm Size:* [82, 111] use the size of a swarm, computed by taking the average distances between drones.

Figure 4.2 shows the measured *Accuracy, Coherence*, and *Swarm Size* values in the area Ⓐ in Figure 4.1. Note that as the mission has obstacles (i.e., obstacles ❸ and ❹), the swarm is expected to be impacted by them. In other words, there should be distinctive changes in the measurement as the reflection of the obstacles' impacts at the area Ⓐ.

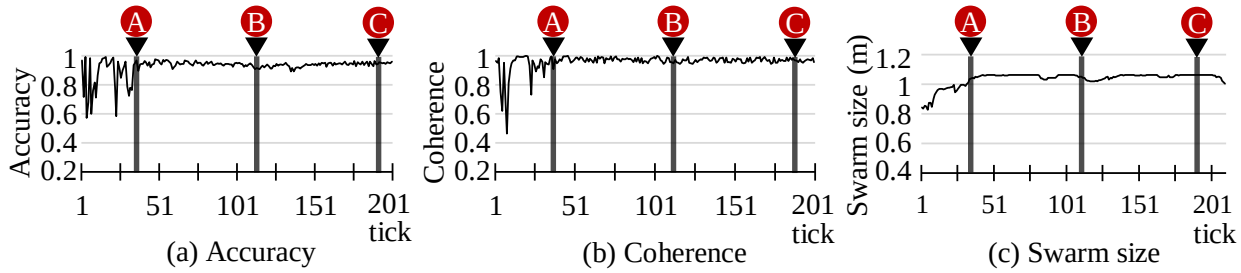


Figure 4.2: Measurements of *Accuracy*, *Coherence*, and *Swarm Size*.

However, as shown in Figure 4.2, the metrics do not show clear distinctions between B and C. Specifically, the accuracy and coherence values depend on the drone's flight directions, which capture only one of the factors affecting the swarm. There are no clear differences between changes caused by the normal flight (before the area A) and flight between narrow passage (the area A). The swarm size, Figure 4.2-(c), also indirectly reflects only one of the factors: the size of the physical space that the swarm goes through. As a result, while the three metrics capture a certain aspect of behaviors, they do not clearly suggest an impact of obstacles (obstacle 3 and 4) and the swarm's interaction with them.

Our Approach. We leverage a metric proposed by [84], called the degree of causal contribution (or Dcc).

Figure 4.3 shows how Dcc captures each of the factors to abstract a drone's behavior. We run multiple executions where in each execution, we eliminate each factor at a time. Any changes (i.e., Δ in Figure 4.3) in the drone's behavior can be attributed to the eliminated factor. For example, the impact of a wall is measured by observing Δ from the run without a wall shown in Figure 4.3-(b). Figure 4.3-(e) is the Dcc value obtained in this experiment, which is a proportionally accumulated Δ values. It conducts such an experiment for all the drones in a swarm and aggregates the values to obtain the Dcc of a swarm, and the details can be found in [170, 84].

Compared to the existing metrics, Dcc *comprehensive* captures diverse causes of the swarm's behaviors. Figure 4.4 shows an example of Dcc measured during the mission from A to C. Each color represents a different factor contributing to the swarm's behavior. Observe that unlike other

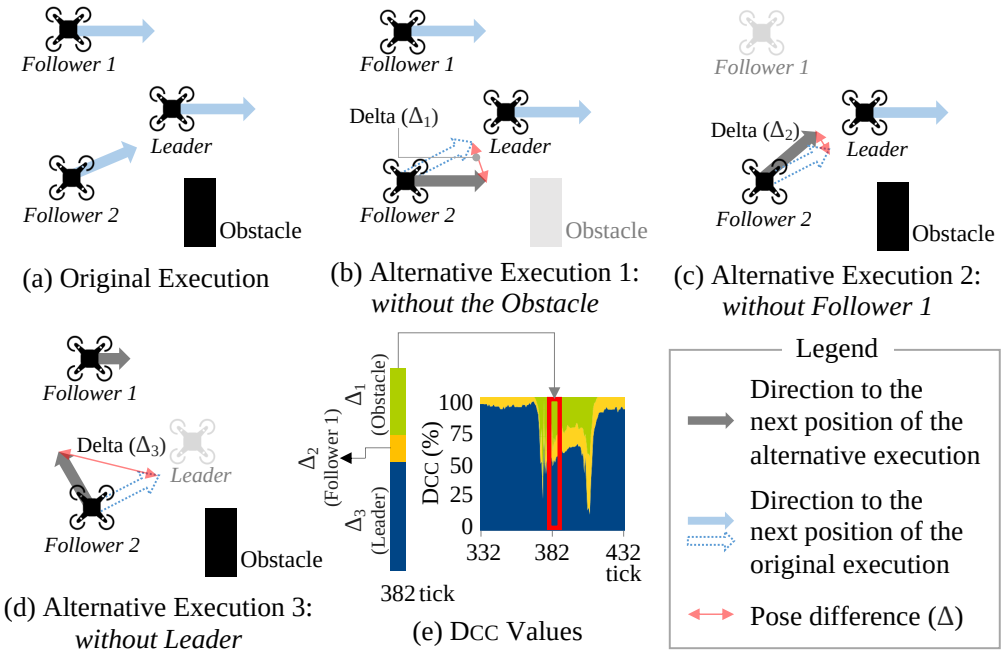


Figure 4.3: Abstracted swarm behaviors in Dcc.

metrics, Dcc shows different value patterns between A∼B and B∼C.

Behavior Patterns Reflecting Complexity of Environment: From the Dcc values as shown in Figure 4.4, we recognize distinctive swarm's behaviors by identifying patterns of the Dcc values such as an appearance/disappearance of factors in Dcc values or changes in values' trend (i.e., increasing or decreasing). From the recognized Dcc patterns, we observe the significant impact of the obstacle (black color), while the other methods cannot do as well. Specifically, obstacle ❷ is reflected in A∼B and obstacle ❸ and ❹ are reflected in B∼C. Observe that the more significant impact is represented as larger values that increase rapidly. Intuitively, more drastic changes of Dcc indicate a complex environment that imposes more interactions on the swarm. For example, the area between A and B is less complex than the area between B and C, which is reflected in Dcc. Details of how we calculate the changes of values and how to use this as feedback for the mutation are in Section 4.3.3.

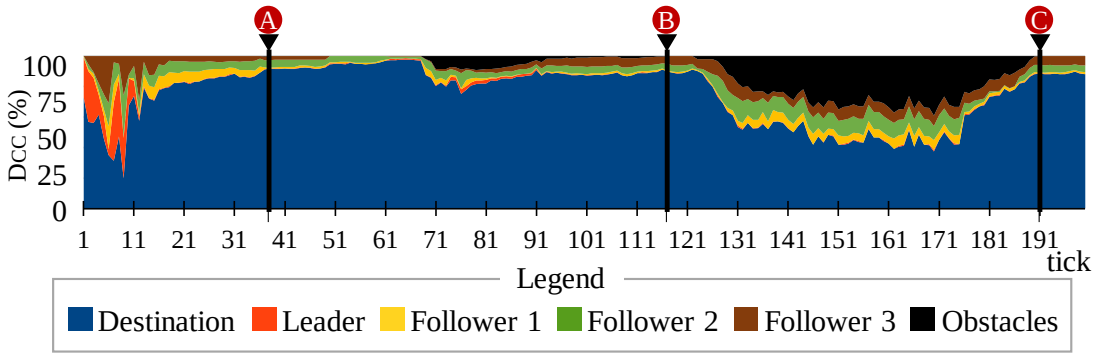


Figure 4.4: Abstraction of swarm behavior represented by Dcc (*Destination* indicates the swarm's causal impact of flying towards to the goal. *Leader* and *Follower 1∼3* are the impact of individual drones. *Obstacles* represents the impact of physical objects to the swarm's behavior).

4.3 Design

We describe the proposed approach that generates a more challenging environment for a swarm in order to detect corner cases more effectively. The overall procedure is shown in Figure 4.5. It takes two inputs: a target swarm algorithm and a swarm mission. It runs an initial test with the given input environment. Generated Dcc is analyzed to understand the complexity of the environment. Based on the complexity score, it mutates the current environment and continues testing until there is no mutatable environmental factor. Finally, while it is not part of this approach, the final mutated environment can be used to further fuzz testing or real-world experiment.

4.3.1 Test Execution

It has a target swarm algorithm instrumented for Dcc and takes a swarm mission including an initial environment. It runs the swarm mission and outputs the mission result and Dcc. After it takes the mutated environment, the current environment is replaced with the mutated one. Upon detecting unexpected behaviors such as such as collision and stopped drone (i.e., immobility of drone), the incident is reported (stored), and the simulation is terminated immediately after logging all swarm states for a later inspection. In other words, a failed mission can be analyzed as a bug caused by the current environment but not used for the mutation process since its Dcc is extremely different from the normal run. Dcc analyzer does not consider this case as a more complex environment.

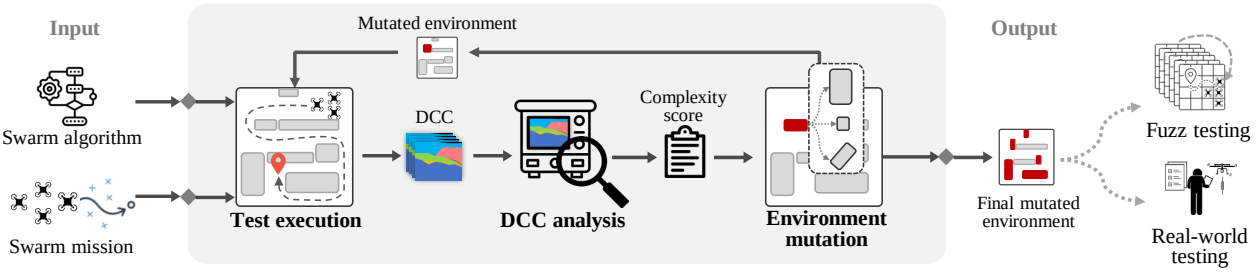


Figure 4.5: Overview of the proposed approach.

4.3.2 Dcc Analysis

In each run, Dcc is obtained, and it analyzes Dcc to determine whether the current environment has become more complex or not by comparing the Dcc from the current and previous runs. The sudden and significant fluctuation in Dcc indicates the rapid change of interaction between the swarm and the environment. We consider the environment for a swarm to be more complex when (1) a swarm has more interactions with environmental factors and (2) its interaction changes rapidly. This is because not every environmental factor (e.g., obstacle) impacts the swarm behavior even when they are placed complexly. Hence, measuring their geographic complexity without considering the impact on the swarm's behavior is not effective. In addition, while environmental factors impact the swarm's behavior, if the impact is weak and stable, it is not helpful to derive unexpected behaviors.

Algorithm for Dcc Computation. Algorithm 3 shows the algorithm to compute Dcc values for the swarm. For each drone (d), Dcc values are computed in a `for` loop from line 3 to line 18.

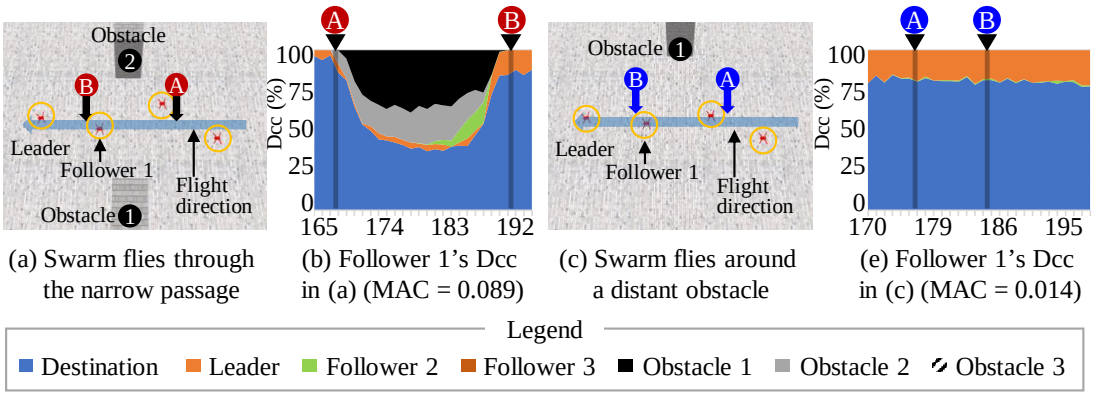
The multiple runs are conducted with perturbations that remove one of the obstacles (O_e), and the neighbor drones (D) as shown in Lines 9∼14. In particular, P_{org} (line 7) represents the original pose of a drone (i.e., without perturbation) and P_i (line 11) is the pose of a drone with an environment in which one (o_i) of the obstacles is removed (i.e., with perturbation). It computes the distance between these two different poses and represents it as Δi. Note that the obstacle set (O_e) changes dynamically depending on the applied mutation strategy, such as inserting or deleting (Section 4.3.3).

Algorithm 3 Dcc computation

Input : D: a set of variables representing drones.
O_e: a set of variables representing objects in the mission environment.
T_{end}: the maximum time limit for the execution.
Output: Dcc: the degree of causal contribution (Dcc) values for swarm mission.

```
1  procedure SwarmDcc(D, O_e, T_end)
2      t ← 0
3      while t ≠ T_end do
           // Each drone d
4          for d ∈ D_v do
5              ΔTotal ← 0
6              O_all ← D ∪ O_e
7              P_org ← GetPose (d, O_all, t) // Pose of a drone d at t
               // Each variable o representing objects including 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
```

Complexity Score. We leverage Dcc as a metric to measure the complexity of the environment (i.e., complexity score) by focusing on the abrupt changes in Dcc. For example, Dcc shows the rapid change (Figure 4.6-(b)) between A and B in the narrow passage (Figure 4.6-(a)) while it shows stable values between A and B (Figure 4.6-(d)) in the simple environment (Figure 4.6-(c)).



(a) Swarm flies through the narrow passage
(b) Follower 1's Dcc in (a) (MAC = 0.089)
(c) Swarm flies around a distant obstacle
(e) Follower 1's Dcc in (c) (MAC = 0.014)

Figure 4.6: Example of different complexity reflected in Dcc.

To measure the rapid change of Dcc, we use the mean absolute change (MAC) [60]. The value range is 0 to 1, and a value close to 1 indicates that the current run has significant changes, while a

value close to 0 implies little changes. For example, Figure 4.6-(d) and (e) (MAC = 0.089 and 0.014, respectively) shows Figure 4.6-(a) is more complex than (b). When the current environment's MAC score is increased 5% more than the previous one, we consider the current environment to be more complex than the previous one. Note that this threshold is configurable.

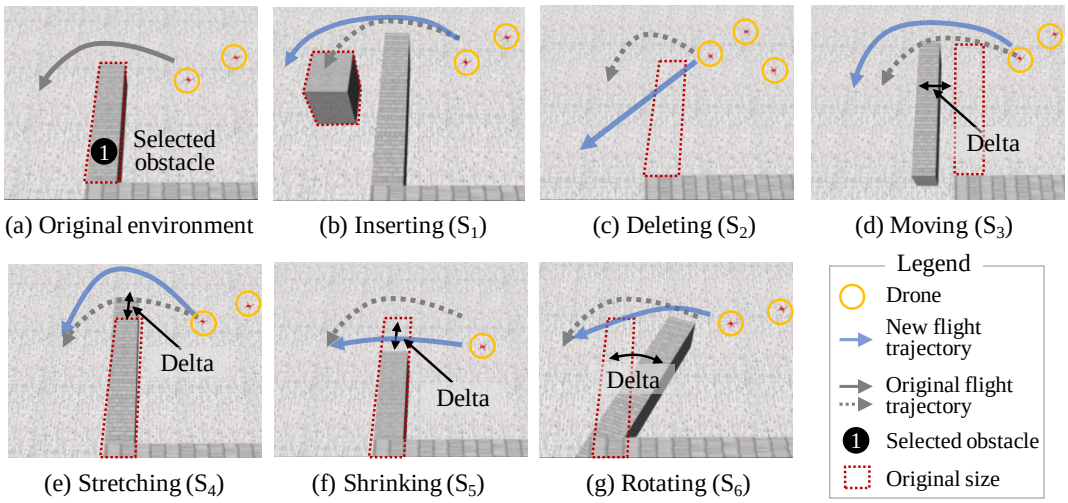
When one execution is twice longer than the average mission duration, we consider it buggy behavior (e.g., straggler or delayed mission). We then store it and analyze it manually after the entire mutation process is over. In this case, we discard the current mutation that causes a longer or a stopped mission. When the crash occurs in the early of the mission, execution time may be shorter than half of the average mission duration. In this case, we run the current execution at most ten times. If all ten executions have crashes, we store the current environment for further analysis and discard the current mutation. Otherwise, we calculate the complexity score and process the next step.

4.3.3 Environment Mutation

This process aims to increase the mutated environment's complexity on the swarm. Specifically, it determines whether to mutate or not based on the complexity score from Dcc analysis. If the current environment is more complex than the previous one, it continues using the most recent mutation strategy. Then, if the current environment has no improvement or becomes less complex, it tries the next mutation strategy.

Mutation strategy. We expect the environmental factor set (e.g., obstacle set) to be provided by users. The strategy selects one obstacle from the set of obstacles in sequential order and applies one of the strategies. Figure 4.7 shows examples of the environment in which each mutation strategy is applied to the original environment (Figure 4.7-(a)). The mutation strategies are as follows:

1. *Inserting (S_1):* It inserts an obstacle at a random place in the target environment as shown in Figure 4.7-(b). The size of the new obstacle is predefined and configurable. In an environment, this strategy typically makes a swarm behavior more complex.



(a) Original environment (b) Inserting (S_1) (c) Deleting (S_2) (d) Moving (S_3)

(e) Stretching (S_4) (f) Shrinking (S_5) (g) Rotating (S_6)

Figure 4.7: Mutation strategies.

2. *Deleting (S_2):* It selects an obstacle randomly and deletes the selected obstacle from the target environment as shown in Figure 4.7-(c). This strategy typically makes a swarm behavior less complex.

3. *Moving (S_3):* It moves a randomly selected obstacle with a predefined distance in one of four cardinal (north, south, east, west) and vertical directions (up and down) as shown in Figure 4.7-(d).

4. *Stretching (S_4):* The selected obstacle is stretched 10% more than the original size in one of four cardinal (north, south, east, west) and vertical directions (up and down). For example, by applying S_4 to the original obstacle in the north direction, the stretched obstacles can be observed as shown in Figure 4.7-(e).

5. *Shrinking (S_5):* It decreases 10% of the size from the original size in one of four cardinal and vertical directions. For instance, changed swarm behavior caused by a shrunk obstacle is observed in Figure 4.7-(f).

6. *Rotating (S_6):* The selected obstacle is rotated by 30 degrees in the pitch, yaw, or roll. For example, Figure 4.7-(g) shows the mutated obstacle after applying S_6. Such change also impacts the swarm's behavior, which is represented by Dcc.

This process tries one strategy with all directions (or degree units) one by one, then moves to the next strategy. The strategies are applied to all obstacles until there is no further increase in complexity. Note that all parameters are configurable.

Mutation Constraints. During the mutating process, mutated obstacles may cause side effects that go against the purpose of the test: (1) it blocks the swarm directly, and (2) it causes meaningless mutation process. To prevent this, we set the mutation constraints that we avoid to mutate in the following areas:

1. *Around the goal:* The goal is directly blocked by mutated obstacles (e.g., inserted or stretched obstacles). For example, if the newly inserted obstacle is spawned at the goal's location, a swarm cannot reach the goal due to the obstacle. To avoid this case, we set the area to avoid to include the area around the goal (e.g., 0.3m from the goal).

2. *Around starting point:* In addition, if the mutated obstacle blocks the starting point, a drone is spawned in the obstacle or close to the obstacle, resulting in an immediate crash. To prevent this, the mutated obstacle should avoid the area around the starting point of the swarm, considering the safety distance (e.g., 0.3m from the starting point).

3. *Inside of obstacles:* This causes the overlapped obstacles. For example, if the inserted obstacle is spawned in the already existing obstacles, it is unnecessary because it does not improve the complexity. To avoid this, the area to avoid includes existing current obstacles.

4. *Outside of the mission area:* The obstacle does not need to be inserted or stretched outside of the mission area because the swarm cannot reach there. Thus, we do not apply Inserting (S_1) or Stretching (S_4) to this area.

In addition, mutated obstacles may impact the main mission route, which changes the goal of the mission itself. For example, if obstacle ❶ in Figure 4.1 is removed by Deleting (S_2), the entire route is directly connected from the starting point to the goal. In this case, this mutation is not selected because it does not improve the complexity of the environment, but we need to make sure the main

mission route is maintained. To do this, we set the fixed obstacles that we do not mutate. This setting is reasonable because there are obstacles, such as the cliffs or huge rocks, that we cannot modify in nature.

Algorithm 4 Generating complex environment

```
Input  : D: a set of variables representing drones.
         O_e: a set of variables representing objects in the mission environment.
         T_timeout: the maximum time limit for the generating (i.e., timeout).
Output: E_comp: a complex environment for swarm testing.
1  procedure GenEnv(D, O_e, T_timeout)
2      O_prev ← O_e // Initial set of objects
3      O_cur ← O_e // Initial set of objects
4      C_noimp ← O_e // Counter for the number of mutations without improvement
5      C_threshold ← 26 // Maximum trials of mutations when no improvement (configurable)
6      Dcc_prev ← 0
7      while the elapsed time of generating did not reach T_timeout do
           // Run a test with the current environment. If the current mission fails, store it for analysis
8          if RunSwarm(E_cur) = MISSION_FAILURE then
9              E_failed ← E_failed ∪ E_cur
           // Obtain Dcc values with the current environment
10         Dcc_cur ← SwarmDcc(D, O_e, T_end)
           // Check whether the current environment becomes more complex than before
11         IsComplexDcc ← False
12         for r ∈ D do
13             if GetMAC(Dcc_cur(r)) > GetMAC(Dcc_prev(r)) then
14                 IsComplexDcc ← True
15         if IsComplexDcc = True then
16             Dcc_prev ← Dcc_cur
17             O_prev ← O_cur
18             E_comp ← O_cur
19             O_cur ← MutateEnv(O_cur, δ) // Mutate the current environment using the same mutation strategy
20             C_noimp ← 0
21         else
               // Terminate if there is no improvement.
22             if C_noimp > C_threshold then
23                 break
24             O_cur ← MutateTest(O_prev, ℝ) // Mutate the previous environment using the next mutation strategy
25             C_noimp ← C_noimp + 1
26     return E_comp
```

Algorithm. Algorithm 4 shows the algorithm for the overall procedure of generating a complex environment. `GenEnv()` describes the entire generating a complex environment process, including comparing complexity score and mutation. The algorithm takes three inputs: (1) D: a set of variables representing drones, (2) O_e: a set of variables representing objects in the mission environment, and (3) $T_{timeout}$: the maximum time limit for the generating. While generating environments, if the current run fails (e.g., crash), we store it for analysis (line 8∼9). In this case, Dcc is extremely different from the normal run (i.e., shorter than the normal run), which is calculated as low MAC

values in `GetMAC()` and discard the current environment (line 21∼25). When the MAC value of the current Dcc is larger than the previous MAC values, we consider the current environment becomes more complex than the previous one (line 12∼14). We then update the previous run's values (Dcc_{prev}, and O_{prev}) and mutate the current environment using the same mutation strategy (line 15∼20). Otherwise, we discard the current environment, meaning that we mutate the previous environment using the next mutation strategy, as the previous mutation strategy was ineffective. If this case happens over time, we terminate the mutation because every possible mutation strategy is ineffective, meaning there is no more thing to mutate (line 22∼23). `MutateEnv()` (line 19 and 24 in Algorithm 4) includes mutation constraints. Note that configuring mutation constraints, including the area to avoid, does not require any domain knowledge. We expect the users to provide this configuration before the testing as they are related to the details of the mission (e.g., coordinates of starting point or goal).

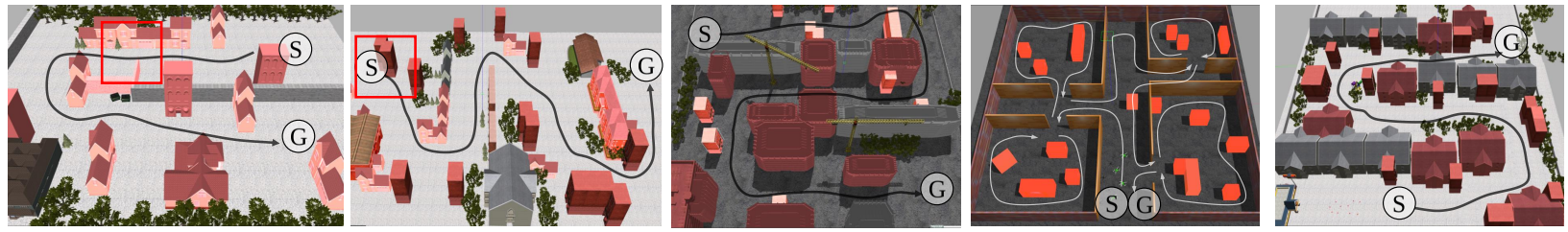
4.4 Evaluation

4.4.1 Experiment Setup

Swarm Algorithms. We use 4 swarm algorithms shown in Table 4.1. To select the algorithms, we search GitHub repositories containing drone swarm simulations from 2010 to 2023, listing 96 algorithms. Among these, we select 4 swarm algorithms that have (1) collective swarm behaviors (e.g., controlling drones as a swarm), (2) a path to follow to the destination (e.g., waypoints), and (3) interaction between drones and environmental factors (e.g., avoiding obstacles). More details of the selection process can be found in [171].

A1. Adaptive Swarm [2] conducts a navigation mission from the starting point to the predefined destination (i.e., goal) while maintaining a formation and avoiding obstacles. It supports a number of drones in a swarm for up to 20.

A2. Swarmlab [164] aims to move a swarm from the current position to a goal, avoiding obstacles.



(a) A1-M1 Flying along the wall (b) A1-M2 Flying through houses (c) A1-M3 Flying through high-rise buildings (d) A1-M4 Indoor rescue support mission (e) A2-M1 Flying through apartment town

(f) A2-M2 Flying through downtown (g) A3-M1 Flying through factory site (h) A3-M2 Disaster rescue support mission (i) A4-M1 Environment monitoring (j) A4-M2 Indoor mapping

Figure 4.8: Mission visualization (via Gazebo simulator [3]) and mutated obstacles (marked as red color).

Table 4.1: Selected Algorithms for Evaluation

ID	Name	SLOC	Drones	Objective
A1	Adaptive Swarm [2]	3,091	4	Flight avoiding static & dynamic obst.
A2	Swarmlab [164]	13,213	10	Flight avoiding static obstacle
A3	MAS simulation [117]	3,795	10	Flight avoiding static obstacle
A4	Zhou's [209]	2,951	5	Flight avoiding static & dynamic obst.

It tries to match the speed with other drones during the mission.

A3. MAS simulation [117] aims to simulate the flocking behavior of drones in a swarm with obstacles that need to be avoided. Individual drones actively interact with neighbor drones, allowing drones to join and leave a swarm.

A4. Zhou's swarm algorithm [209] conducts a cooperative navigating mission using multiple drones. The swarm can dynamically change the formation at runtime, avoiding obstacles under flight time constraints.

Evaluated Missions. We use 10 missions (A1-M1∼A4-M2[1]) for our evaluation to cover various flying scenarios as shown in Figure 4.8. Descriptions for the missions are as follows:

- *M1-A1:* The swarm flies through an open field along the wall.
- *M1-A2:* The swarm passes through houses along the 'S' shaped path to reach the goal.
- *M1-A3:* The swarm conducts a construction site monitoring mission.
- *M1-A4:* The swarm conducts search and rescue missions for indoor victims.
- *M2-A1:* The swarm conducts a delivery mission flying through the apartments.
- *M2-A2:* The swarm conducts a delivery mission flying through the downtown.
- *M3-A1:* The swarm flies through the factory site to monitor around the main facility.
- *M3-A2:* The swarm searches for victims and delivers supplies to the destroyed town.
- *M4-A1:* The swarm conducts an environment monitoring mission on the mountain path.
- *M4-A2:* The swarm conducts mapping and monitoring missions in the unorganized inside of a warehouse.

[1] Ai-Mj means the ith algorithm's jth mission.

Implementation. We implement prototypes of our approach in the programming language that the original algorithm is written in Python and Matlab. Our implementation includes modifications of existing simulators. To this end, we write 878, 650, 520, and 483 lines for implementing our approach in Python (A1) and Matlab (A2, A3, and A4), respectively.

Experiment Setup. All experiments are performed on a machine with an Intel Core i9 3.70GHz processor and 64GB RAM, running Ubuntu 22.04. We use five Crazyflies [23] drones for the real-world experiment in Section 4.4.5.

Table 4.2: Mutation Parameters Used in Our Evaluation

ID	Size (S_1)[1]	Mov. dist. (S_3)[2]	Inc. dist. (S_4)[3]	Dec. dist. (S_5)[4]	Rot. ang. (S_6)[5]
A1	0.2 m	0.2 m	0.1 m	0.1 m	30°
A2	1.0 m	0.5 m	0.5 m	0.5 m	30°
A3	0.5 m	0.3 m	0.3 m	0.3 m	30°
A4	0.3 m	0.2 m	0.2 m	0.2 m	30°

1: The side of a cube (e.g., a cube with a side of 0.2m), 2: Moving distance, 3: Increasing distance for Stretching, 4: Decreasing distance for Shrinking, 5: Rotation angle (clockwise)

4.4.2 Effectiveness of Mutated Environments

Table 4.2 shows the parameters used in the mutation process. We configure the value of parameters considering not the size of the mission area (i.e., the original environment) but the size of the drone (i.e., the configuration of the algorithm), which can be found easily in a configuration file or documentation. For example, the default size of the drone of A2 is bigger than A1. Hence, most of the parameters are configured bigger. In addition, this determines the granularity of the mutation. For example, when an obstacle is inserted (i.e., S_1), it is difficult to capture the effectiveness of the inserted obstacle if the size is too small. On the other hand, if the size is too big, the value of Dcc changes significantly (i.e., the effectiveness of the inserted obstacle is captured easily). As a result, the mutation process is converges quickly as mutated obstacles take more space.

Figure 4.8 shows the mutated environment that our proposed approach generates. We conduct Dcc

guided fuzz testing [166] on each mutated environment for 24 hours to understand the effectiveness of generated environments. To this end, we compare the result of fuzz testing with the mutated environment and with the original environment (i.e., without the mutated environment).

Table 4.3 shows the statistics of the environment used for fuzz testing and results. We also observe 44 more unique swarm behaviors, including 13 more buggy behaviors that were not observed during 24 hours of fuzz testing with the original environment. The second and seventh columns indicate the time taken for the missions. Note that there is no case that mutated obstacles directly block the swarm (Section 4.3.3), and the buggy behaviors that the swarm does not process (e.g., straggler case) are excluded. We set the timeout to be two times the mission duration. For example, when the swarm cannot reach the goal until the 800 ticks for A1-M1, we terminate the execution and consider this a buggy behavior (i.e., straggler). The values of the seventh column are increased because the swarm's route is increased because of more interactions with added obstacles. The number of obstacles increases for the same reason (the third and eighth columns). This indicates not the number of mutated obstacles but the maximum number of obstacles that make the environment the most complex. The trend of the complexity score follows along with it (the fourth and ninth columns). Environments that have more increased values for the number of obstacles and higher complexity scores have more space to place additional obstacles (e.g., A1-M3 or A3-M1). On the other hand, although the size of the original environment is large, the increase is not large when there is not enough space to add obstacles (e.g., A1-M4 or A4-M1). The fifth and tenth columns indicate the number of unique behaviors. We consider that the behavior is unique when the Dcc value differs by more than 10% from the others. This indicates using mutated environments induces more diverse behaviors than the original environments as well as the original environment is not complex enough. In addition, the sixth and eleventh columns show the number of bugs. We analyze them manually and differentiate them as different bugs.

Except for overlapped bugs from the mutated environment, a total of 5, 3, 2, and 3 more bugs are observed than the original environment for A1, A2, A3, and A4, respectively. We manually analyze the observed unique behaviors and bugs and find that those from the original environment

are the subset of the results from mutated environments. We present the details of three new bugs in Section 4.4.5. The details of other bugs can be found on [171].

4.4.3 Effectiveness of Dcc

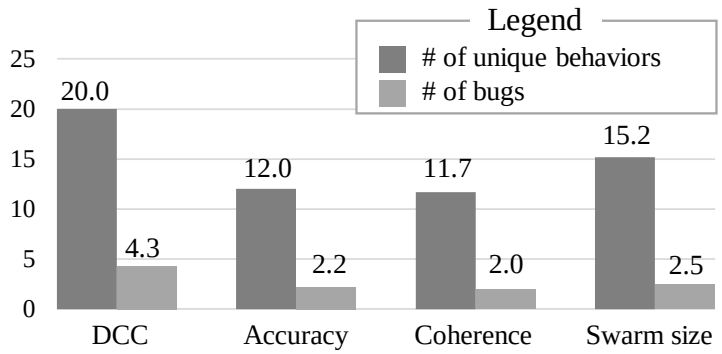


Figure 4.9: The average number of unique behaviors and bugs from the environments mutated by Dcc, Accuracy, Coherence and Swarm size.

To understand the effectiveness of Dcc, we compare the number of unique behaviors and bugs found using the environment mutated by Dcc, Accuracy [5, 178], Coherence [185, 25] and swarm size [82, 111]. The result shows using Dcc outperforms the others in Figure 4.9. This is because Accuracy and Coherence are heading angle-based metrics, meaning that they are relatively difficult to generate meaningful obstacles as the degree of change from obstacles is smaller than the degree of change in the existing route, such as an S-shaped route, Swarm size is also relatively less affected by obstacles. For example, unless an extremely narrow passageway is created and it greatly distorts the swarm, the swarm size does not change significantly because there is repulsion between drones (i.e., a tendency to maintain the formation and safety distance).

4.4.4 Trend of Complexity Score

We measure the changes in complexity score while mutating. Figure 4.10 shows the trend of changed complexity score of each mutation process in A1. The mutations for A1-M1, A1-M2, A1-M3, and A1-M4 converge around 5.2 hours, meaning additional mutations from this point do not add any

Table 4.3: Results of Fuzz Testing

ID	Without mutated environments					With mutated environments					
	Mission dur.[1]	# of obs.[2]	Compl. score[3]	Uniq.[4]	# of bugs	Mission duration	# of obs.	Fixed obs.[5]	Compl. score	Unique	# of bugs
A1-M1	382	17	0.021	8	1	424 (+11%)	33 (+16)	3	0.062 (+0.041)	20 (+12)	5 (+4)
A1-M2	341	23	0.031	5	1	395 (+16%)	36 (+13)	5	0.071 (+0.040)	22 (+17)	5 (+4)
A1-M3	322	18	0.025	10	3	355 (+10%)	26 (+ 8)	3	0.082 (+0.057)	19 (+ 9)	4 (+2)
A1-M4	588	16	0.034	4	3	628 (+14%)	23 (+ 7)	1	0.088 (+0.064)	20 (+16)	4 (+1)
A2-M1	2,866	12	0.028	2	3	3,192 (+11%)	29 (+17)	4	0.079 (+0.051)	15 (+13)	5 (+2)
A2-M2	3,404	14	0.031	3	1	3,812 (+ 9%)	22 (+17)	5	0.075 (+0.044)	18 (+15)	4 (+3)
A3-M1	552	7	0.019	5	2	603 (+13%)	22 (+19)	3	0.080 (+0.061)	18 (+13)	4 (+2)
A3-M2	448	8	0.018	7	3	554 (+14%)	20 (+12)	4	0.073 (+0.055)	19 (+12)	5 (+2)
A4-M1	126	7	0.025	11	1	149 (+18%)	15 (+ 8)	3	0.069 (+0.044)	29 (+18)	4 (+1)
A4-M2	118	8	0.029	8	1	141 (+20%)	16 (+ 8)	4	0.062 (+0.033)	20 (+12)	3 (+2)

1: Time taken to execute the mission, 2: The number of obstacles in the mission environment, 3: Complexity score, 4: The number of unique behavior patterns found in 24 hours, 5: The number of fixed obstacles (i.e., the exception of mutation)

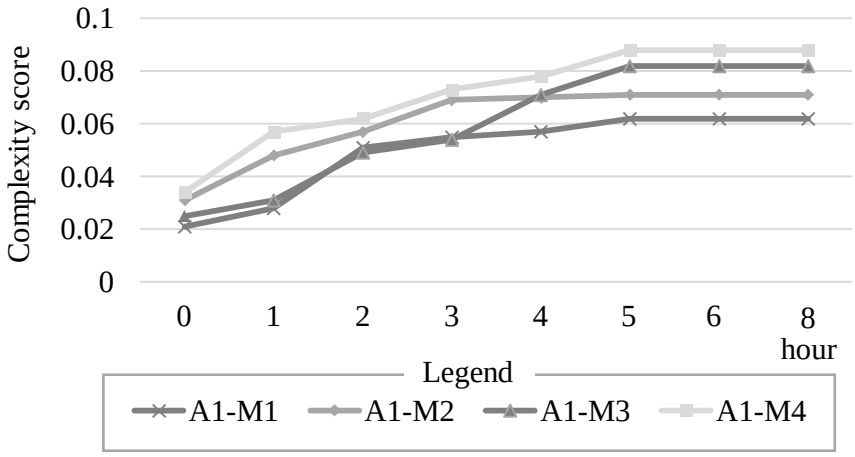


Figure 4.10: The complexity score of four missions of A1 over time.

improvement. This shows that the final mutated environments from our proposed approach are the most complex. The results of the other algorithms can be found in Section A.10 or [171]. We also observe that the converged score depends not on the algorithm but on the space for the drone's possible path. This is because more space allows the flexibility for a more complex environment. For example, A2-M2, which consists of narrow passages, does not have enough space to have more mutated obstacles and shows a short time to converge. In addition, it is related to the average execution time. For example, A1-M4 has a relatively longer execution time than A1-M2, which takes 2.2 more hours, as shown in Figure 4.10.

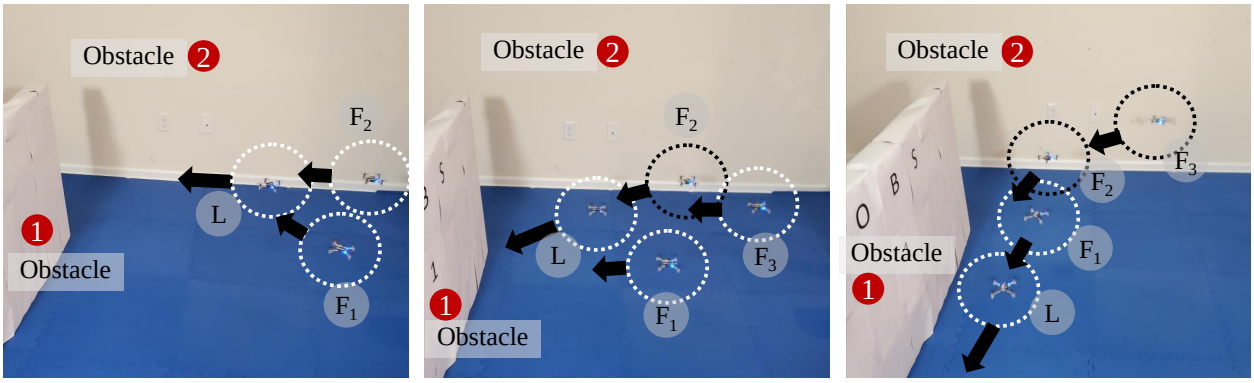
4.4.5 Case Study

Wrong update of waypoints

We found a new buggy behavior in the red box of Figure 4.8-(a) and reproduced this case using real-world drones in our lab as shown in Figure 4.11. In this bug, the entire swarm does not pass over the obstacle (❶), though the passage between the obstacle ❶ and ❷ is large enough. This is because the waypoint is updated in the wrong way. Specifically, in this algorithm, the waypoint is updated to the next waypoint when the leader drone is close (less than 0.8 meters) to the current

waypoint.

In this case, due to the thin (mutated) obstacle 1, the waypoint is updated to the next one when the leader drone approaches the current waypoint even though it does not go over the obstacle 1 yet (Figure 4.11-(a)). As the leader drone is heading to the next waypoint, located south over the obstacle 1 as shown in Figure 4.11-(b), the entire swarm is left behind the wall (Figure 4.11-(c)). We change the waypoint algorithm (10 SLOC) to update the waypoint after the leader drone goes over the obstacle to fix the bug.



(a) Swarm approaches to the wall. (b) Swarm does not go over the obstacle. (c) Swarm is left behind the wall.

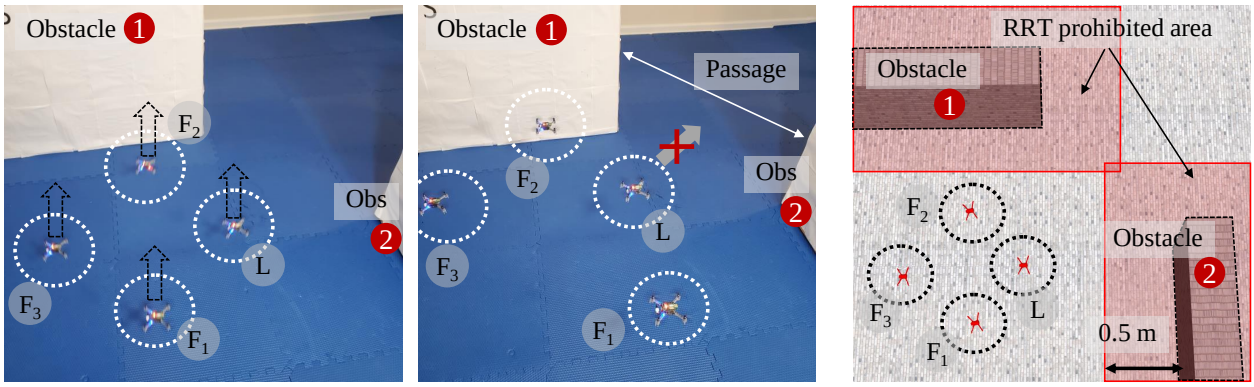
Figure 4.11: Drones cannot go over the obstacle due to the wrong update of waypoints caused by the thin obstacle.

Infinite Routes Exploring

Figure 4.12 shows three screenshots of a failed mission due to the bug we observed when using a mutated environment (red box area in Figure 4.8-(b)), which we reproduced in the lab with real-world drones. This case is what we found in the middle of the mutation process, in which the mutated obstacle blocks the swarm, and we stored this for further analysis. In this swarm algorithm, to generate waypoints, the global planner leverages Rapidly-exploring Random Tree (RRT) [101] algorithm. When obstacles are spawned by S_1 close to the starting point as shown in Figure 4.12-(a), RRT algorithm cannot explore the feasible route for the swarm, even though the space between mutated (i.e., inserted) obstacles (e.g., obstacle 1 and 2 in Figure 4.12) is enough

to pass for swarm. Note that these obstacles are not spawned in the area to avoid (Section 4.3.3). As a result, drones in a swarm takeoff (Figure 4.12-(a)) but do not move (Figure 4.12-(b)).

This is because the RRT algorithm cannot explore the space if the space is close to the obstacles, and the parameter for this is set bigger than the safety distance (i.e., a possible path for drones). Therefore, the space where the path search begins and the space of the goal are disconnected by the red area (i.e., prohibited area) of Figure 4.12-(c). After we fix this parameter from 0.5 to 0.3, we observe RRT finds a feasible route as long as the obstacles are mutated under our constraints.



(a) Drones takeoff.

(b) The passage is wide, but swarm does not move.

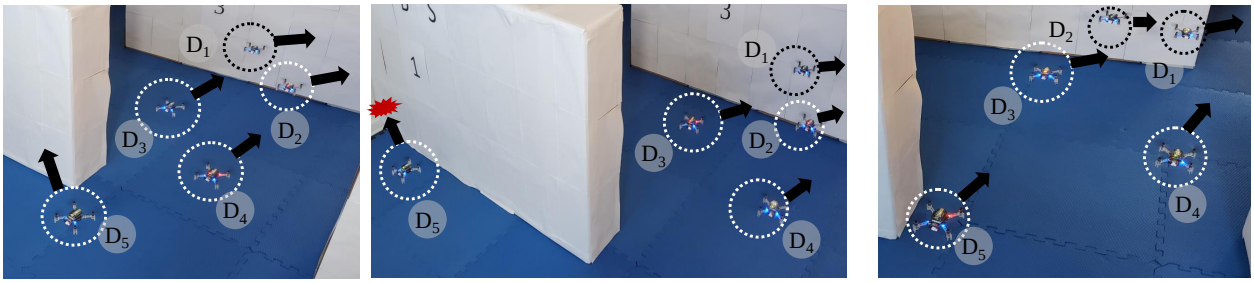
(c) The Route is not generated because of wrong configuration.

Figure 4.12: Drones do not move after takeoff because the route is not generated.

Detached Drone from Swarm

Figure 4.13 shows that the buggy behavior in A3-M1 is reproduced with real-world drones. In A4-M2, one drone flies along the wall which is in the wrong direction after being blocked by other drones. The swarm behaviors from the mutated environment and the original environment are shown in Figure 4.13-(a)∼(b) and Figure 4.13-(c), respectively. The swarm (D_1∼D_5) enters the narrow passage that consists of obstacles. Specifically, D_5 is blocked by D_3 and D_4, then D_5 moves toward the west along the wall, which is stretched by our approach. Once it happens, D_5 tries to detour the obstacle through the north, which is unfortunately blocked by a wall, hence crashing at the end (Figure 4.13-(b)).

We manually analyze the root cause and observe that the coefficient for the repulsive force is not big enough. After we modify it to a bigger value (e.g., 0.6 to 0.8), even when the other neighbor drones block the drone (D_5), it does not go toward the obstacles, not leading to a crash. On the other hand, Figure 4.13-(c) shows that the buggy behavior is not shown at the same place in the original environment because the width of the original passage is wide enough, and it cannot cause the crash.



(a) D_5 lies in the wrong direction after being blocked by D_3 and D_4.

(b) D_5 detaches from the swarm and crashes into the wall. (Mutated environment)

(c) Swarm passes the passage safely. (Original environment)

Figure 4.13: One drone is detached from the swarm because of the narrow passage (mutated obstacle).

4.5 Discussion

Additional Mutation Strategies. There can be more advanced mutation strategies, which may improve SwarmGen's performance. One can add a new mutation strategy by defining how to change obstacles. Applying a combination or a different order of mutations also can be possible. The essence of this research is to show the feasibility of Dcc based environment mutation approach.

Overhead. The proposed approach runs a number of tests during the mutation process to improve the complexity of the environment. Note that the process of generating a complex environment is done offline. The overhead of computing Dcc and comparing the complexity score of the environment at runtime is less than 8%. Fuzz testing using the mutated environments (i.e., the final output) does not require any instrumentation.

Scalability and Usability. The design of the proposed approach is general but it requires some

engineering effort to apply it to other swarm algorithms. In particular, two tasks are required to support a new swarm algorithm: (1) instrumenting the algorithm to integrate the proposed approach (e.g., 334 SLOC for Adaptive Swarm), (2) identifying parameters for the mutation (e.g., the basic size of the inserted obstacle as shown in Table 4.2). In this research, to complete two tasks for an algorithm, it took 12∼20 hours by a graduate student with moderate experience in swarm algorithms. Note that this effort is required one time for each algorithm.

Future Direction. We consider future direction from two aspects. First, for the empirical aspect, future work can include applying the proposed approach to diverse algorithms for multi-agent systems (e.g., multiple autonomous driving systems) and analyzing their effectiveness. Second, for the technical aspect, the part of current processing that requires user-defined values (e.g., the default size of obstacles to insert in S_1) can be automated. In addition, expanding the mutation range to the other environmental factors (e.g., weather or temperatures) and combination with the existing state-of-the-art techniques (e.g., introducing attack drones to perturb the swarm behaviors more [84]) can be a promising way to reveal more unique behaviors and bugs.

4.6 Related Work

Simulation-based Testing for Robotics. Due to the high cost of field testing, simulation-based approaches have been common alternatives [3, 159, 9, 76, 1]. [5, 208, 4] test swarm's collective motion using simulation before the field test. Specifically, [5] proposed a decentralized flocking approach using potential field models focusing on the application of algorithms rather than the improvement of the environment. [208] proposed swarm algorithms that adapt under unknown clutter environment, while the environment is already defined by the authors. [4] shows their proposed approach with a constrained environment, which has high obstacle density. However, their research focuses on using onboard sensor data rather than making a more complex outdoor environment. Our work is complementary to them as our proposed approach can provide the most complex environment for them.

To test the robotics algorithm using simulation more effectively, several techniques that modify the given environment are proposed. There exist checkpointing techniques [95, 74, 157] that can store a specific state of the system and replay it. They use the sliding window to decide the restore point but it is manually defined concerning the event of interest (e.g., failures). As a result, their effectiveness is dependent on the quality of the restore point, which requires domain knowledge from the experts. Unlike these studies, our proposed work analyzes the complexity of environments by leveraging Dcc without requiring domain knowledge. In addition, our work is complementary to these studies as Dcc provides a potential restore point for the event of interest as a quantitative measurement.

Generating Test Cases. There is a line of research focused on generating test cases [91, 92, 173, 57, 206] to improve the effectiveness of testing. In particular, test generation approaches based on real-world data using traffic rules/regulations [173], police reports [57], and accident cases [206]. However, these studies are difficult to apply to drone's domain where there are no predefined regulations (e.g., traffic rules of specific regions) that can be used as guidance. [70] generates more stressful test cases by leveraging dynamic physical models of the drone using simulation and Trey et el. propose a fuzzing approach to find failure-inducing input for mobile robots [196]. However, these studies are conducted under a given default environment. Hence, our work is complementary to them as it can provide a more complex environment that can be used as the base for them.

4.7 Summary

In this research, we propose the automated swarm mission environment generation technique based on the analysis of swarm behaviors. In particular, we leverage the Dcc to abstract the swarm behavior to measure the interactions between the swarm and the mission environment. We evaluate our proposed approach using four real-world swarm algorithms and the result shows that the mutated environment from our proposed approach is effective in discovering more unique behaviors and bugs than using the original given environment.

Chapter 5

Conclusion

The rapid advancement of emerging technology significantly improves our lives. Swarm robotics is one of them, which attracts people's attention because of its potential impact on society. At the same time, exhaustively testing a drone swarm algorithm is crucial for obtaining robust drone swarm algorithms. However, testing swarm algorithms becomes more challenging as swarm algorithms and missions become more complex and sophisticated.

This dissertation outlines and tackles challenges associated with testing for robust swarm algorithms. (1) Due to the large configuration space, the complexity of the swarm algorithms, and the complex dependencies make debugging and fixing configuration bugs challenging. (2) Also, existing software testing approaches such as coverage-guided fuzzing are not effective in testing robotics systems because their execution is highly iterative showing quickly covered code and branches. (3) Furthermore, the default mission environments are given by the developers with the swarm algorithms, but they are not enough to discover the corner cases of swarm behaviors, which is required for swarm testing.

We proposed SWARMBUG, a debugging approach for resolving configuration bugs in swarm algorithms (Chapter 2). SWARMBUG automatically identifies the causes of configuration bugs by creating new executions with mutated environment configuration variables. It compares the new executions with the original execution to find the causes of the bug. Then, given the cause, SWARMBUG applies four different strategies to fix the bug by mutating swarm configuration variables, resulting in fixes for the configuration bugs. Our evaluation shows that SWARMBUG is highly effective in finding fixes for diverse configuration bugs in swarm algorithms.

In addition, we develop a novel fuzz testing approach for swarm robotics, SwarmFlawFinder, to discover swarm algorithms' logic flaws (Chapter 3). 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).

Finally, we propose SwarmGen, an automated swarm mission environment generation technique based on the analysis of swarm behaviors (Chapter 4). In particular, we leverage the Dcc to abstract the swarm behavior to measure the interactions between the swarm and the mission environment and use it as the complexity score for the mutation process of the environment. We evaluate our proposed approach using four real-world swarm algorithms. The result shows that the mutated environment from our proposed approach is effective in discovering more unique behaviors and bugs than using the original given environment.

To summarize, this research proposed methods using a novel metric, Dcc, to abstract the swarm behaviors that interact with the environment. By leveraging Dcc, we develop a debugging system, a feedback-guided fuzz testing system, and a generating complex environment system. The source code of the prototype, developed tools, and more data of this work are made available on GitHub for the community [166, 170, 171].

This research can be extended along two dimensions, empirical and technical, using the following ideas. For the empirical aspect, applying the proposed techniques to more diverse swarm algorithms/systems, including ADS (Autonomous Vehicle System), can be the future work. For example, our approaches are applicable to the swarm using ground rovers [169, 176] or autonomous underwater vehicles [139, 182, 106], as long as the user can instrument the target algorithm to obtain Dcc values. In this respect, further analysis of the cost and benefits of the application of our work can be another future work.

In addition, to support the semi-automated process in our work such as identifying key parameters used as inputs in Swarmbug (Section 2.4.1), or configuring the parameter values for the mutation in SwarmGen (Section 4.3.3) can be the future work as the technical aspect. In terms of the scalability

of the proposed approach, it requires some engineering effort, while our approaches are general and applicable to other swarm algorithms. We explain that it took less than 20 hours (by a graduate student with moderate experience in drone swarm algorithms) for the application of each technique. In this respect, further analysis on incorporating user feedback will enhance the performance of our approaches and give more insights for better applicability and scalability. Also, considering Dcc is based on counterfactual executions, our approaches are complementary to the techniques embedded in the agent (i.e., machine learning techniques for better visual perception [58, 205]). For example, the optimization of swarm operation can be achieved by leveraging machine learning techniques (e.g., reinforcement learning [77, 78]), but there still needs to be a metric that can abstract the swarm behavior. The root cause of the discovered buggy behavior can be placed in embedded techniques, but our approaches can assist in finding more corner cases and debugging processes.

Bibliography

[1] Afsoon Afzal et al. “A study on challenges of testing robotic systems”. In: *IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 2020.

[2] Ruslan Agishev. “Adaptive Control of Swarm of Drones for Obstacle Avoidance”. MA thesis. Moscow, Russia: Skolkovo Institute of Science and Technology, 2019.

[3] Carlos Aguero et al. “Inside the Virtual Robotics Challenge: Simulating Real-Time Robotic Disaster Response”. In: *Automation Science and Engineering, IEEE Transactions on* (2015).

[4] Afzal Ahmad et al. “Autonomous aerial swarming in gnss-denied environments with high obstacle density”. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2021, pp. 570–576.

[5] Dario Albani et al. “Distributed Three Dimensional Flocking of Autonomous Drones”. In: *2022 International Conference on Robotics and Automation (ICRA)*. IEEE. 2022, pp. 6904–6911.

[6] *Autonomous UAVs Swarm Mission*. https://github.com/AlexJinlei/Autonomous_UAVs_Swarm_Mission. 2018.

[7] Omar M Alhawi, Mustafa A Mustafa, and Lucas C Cordeiro. “Finding Security Vulnerabilities in Unmanned Aerial Vehicles Using Software Verification”. In: *arXiv preprint arXiv:1906.11488* (2019).

[8] Javier Alonso-Mora, Stuart Baker, and Daniela Rus. “Multi-robot navigation in formation via sequential convex programming”. In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2015, pp. 4634–4641.

[9] Dejanira Araiza-Illan et al. “Systematic and realistic testing in simulation of control code for robots in collaborative human-robot interactions”. In: *Annual Conference Towards Autonomous Robotic Systems*. Springer. 2016, pp. 20–32.

[10] *ArduCopter*. https://ardupilot.org/copter/docs/introduction.html. 2020.

[11] H. Asama et al. "Functional distribution among multiple mobile robots in an autonomous and decentralized robot system". In: *Proceedings. 1991 IEEE International Conference on Robotics and Automation*. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 1991, pp. 1921, 1922, 1923, 1924, 1925, 1926. DOI: 10.1109/ROBOT.1991.131907. URL: https://doi.ieeecomputersociety.org/10.1109/ROBOT.1991.131907.

[12] Mona Attariyan and Jason Flinn. "Automating Configuration Troubleshooting with Dynamic Information Flow Analysis". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 237–250.

[13] Erkin Bahceci, Onur Soysal, and Erol Sahin. "A review: Pattern formation and adaptation in multi-robot systems". In: *Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-03-43* (2003).

[14] Boldizsár Balázs, Gábor Vásárhelyi, and Tamás Vicsek. "Adaptive leadership overcomes persistence–responsivity trade-off in flocking". In: *Journal of the Royal Society Interface* (2020).

[15] Roberto Baldoni et al. "A survey of symbolic execution techniques". In: *ACM Computing Surveys (CSUR)* (2018).

[16] Jan Carlo Barca and Y. Ahmet Sekercioglu. "Swarm robotics reviewed". In: *Robotica* 31.3 (2013), pp. 345–359. DOI: 10.1017/S026357471200032X.

[17] Jan Carlo Barca and Y. Ahmet Sekercioglu. "Swarm robotics reviewed". In: *Robotica* 31.3 (2013), pp. 345–359. DOI: 10.1017/S026357471200032X.

[18] Rakesh Rajan Beck, Abhishek Vijeev, and Vinod Ganapathy. "Privaros: A Framework for Privacy-Compliant Delivery Drones". In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

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

[20] Gerardo Beni. "From swarm intelligence to swarm robotics". In: *International Workshop on Swarm Robotics*. Springer. 2004, pp. 1–9.

[21] Saddek Bensalem et al. "A verifiable and correct-by-construction controller for robot functional levels". In: *arXiv preprint arXiv:1309.0442* (2013).

[22] Robvanden Berg. *Zebro-Search-and-Rescue*. https://github.com/RobvandenBerg/Zebro-Search-and-Rescue. 2020.

[23] bitcraze. *A lightweight, open source flying development platform based on a nano quadcopter*. https://www.bitcraze.io/products/crazyflie-2-1/. 2020.

[24] bitcraze. *A local positioning system*. https://www.bitcraze.io/products/loco-positioning-system/. 2019.

[25] Alexandre Bonnefond, Olivier Simonin, and Isabelle Guérin-Lassous. "Extension of Flocking Models to Environments with Obstacles and Degraded Communications". In: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2021, pp. 9139–9145. DOI: 10.1109/IROS51168.2021.9635944.

[26] Manuele Brambilla et al. "Swarm robotics: a review from the swarm engineering perspective". In: *Swarm Intelligence* 7.1 (2013), pp. 1–41.

[27] Alexandre Santos Brandão and Mário Sarcinelli-Filho. "On the guidance of multiple uav using a centralized formation control scheme and delaunay triangulation". In: *Journal of Intelligent & Robotic Systems* 84.1 (2016), pp. 397–413. DOI: https://doi.org/10.1007/s10846-015-0300-5.

[28] R. Brooks. "A robust layered control system for a mobile robot". In: *IEEE Journal on Robotics and Automation* 2.1 (1986), pp. 14–23. DOI: 10.1109/JRA.1986.1087032.

[29] Gino Brunner. *autonomous-drone*. https://github.com/szebedy/autonomous-drone. 2019.

[30] Alessandro Calò et al. “Simultaneously Searching and Solving Multiple Avoidable Collisions for Testing Autonomous Driving Systems”. In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*. 2020.

[31] Rafael C. Cardoso et al. “Heterogeneous Verification of an Autonomous Curiosity Rover”. In: *NASA Formal Methods*. Springer International Publishing, 2020. ISBN: 978-3-030-55754-6.

[32] Rafael C. Cardoso et al. “Towards Compositional Verification for Modular Robotic Systems”. In: *Electronic Proceedings in Theoretical Computer Science* (2020).

[33] Pietro Carnelli. *SwarmRoboticsSim*. https://github.com/pc0179/SwarmRoboticsSim. 2017.

[34] Jiongyi Chen et al. “IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing”. In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2018.

[35] Yaohui Chen et al. “Savior: Towards bug-driven hybrid testing”. In: *IEEE Symposium on Security and Privacy (SP)*. 2020.

[36] Hongjun Choi et al. “Detecting Attacks Against Robotic Vehicles: A Control Invariant Approach”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2018.

[37] Soon-Jo Chung et al. “A survey on aerial swarm robotics”. In: *IEEE Transactions on Robotics* (2018).

[38] Timothy H Chung. “Offensive swarm-enabled tactics (offset)”. In: DARPA. 2021.

[39] Mario GCA Cimino et al. “Adaptive Exploration of a UAVs Swarm for Distributed Targets Detection and Tracking.” In: *ICPRAM*. 2019.

[40] James Clause, Wanchun Li, and Alessandro Orso. “Dytan: A Generic Dynamic Taint Analysis Framework”. In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. ISSTA ’07. London, United Kingdom: ACM, 2007, pp. 196–206. ISBN: 978-

1-59593-734-6. DOI: 10.1145/1273463.1273490. URL: http://doi.acm.org/10.1145/1273463.1273490.

[41] Charlie Curtsinger and Emery D Berger. "Coz: Finding code that counts with causal profiling". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. 2015, pp. 184–197. DOI: https://doi.org/10.1145/2815400.2815409.

[42] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. "Qlose: Program repair with quantitative objectives". In: *International Conference on Computer Aided Verification*. Springer. 2016, pp. 383–401. DOI: https://doi.org/10.1007/978-3-319-41540-6_21.

[43] Daniel Wollschlaeger. *Analyzes shooting data with respect to group shape, precision, and accuracy*. https://cran.r-project.org/web/packages/shotGroups/index.html. 2020.

[44] DARPA. *OFFensive Swarm-Enabled Tactics (OFFSET)*. https://www.darpa.mil/work-with-us/offensive-swarm-enabled-tactics. 2017.

[45] DARPAtv. *Teams Test Swarm Autonomy in Second Major OFFSET Field Experiment*. https://www.youtube.com/watch?v=ruWC10AW87E. 2019.

[46] Celso De La Cruz and Ricardo Carelli. "Dynamic modeling and centralized formation control of mobile robots". In: *IECON 2006-32nd Annual Conference on IEEE Industrial Electronics*. IEEE. 2006, pp. 3880–3885. DOI: 10.1109/IECON.2006.347299.

[47] Ankush Desai, Shaz Qadeer, and Sanjit A Seshia. "Programming safe robotics systems: Challenges and advances". In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2018, pp. 103–119. DOI: https://doi.org/10.1007/978-3-030-03421-4_8.

[48] Hoang Tung Dinh and Tom 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. ISBN: 978-3-030-49778-1.

[49] Karel Domin, Iraklis Symeonidis, and Eduard Marin. "Security analysis of the drone communication protocol: Fuzzing the MAVLink protocol". In: (2016).

[50] Marco Dorigo et al. "Evolving self-organizing behaviors for a swarm-bot". In: *Autonomous Robots* (2004).

[51] Marco Dorigo et al. "Swarmanoid: a novel concept for the study of heterogeneous robotic swarms". In: *IEEE Robotics & Automation Magazine* 20.4 (2013), pp. 60–71. DOI: 10.1109/MRA.2013.2252996.

[52] Jan Dufek. *Multi-UAV Cooperative Surveillance*. https://github.com/jan-dufek/multi-uav-surveillance. 2019.

[53] Paul Fiterau-Brostean et al. "Analysis of DTLS Implementations Using Protocol State Fuzzing". In: *29th USENIX Security Symposium*. 2020.

[54] David Hambling. *What Are Drone Swarms And Why Does Every Military Suddenly Want One?* https://www.forbes.com/sites/davidhambling/2021/03/01/what-are-drone-swarms-and-why-does-everyone-suddenly-want-one/?sh=2a5f085d2f5c. 2021.

[55] G Matthew Fricke et al. "A distributed deterministic spiral search algorithm for swarms". In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016.

[56] Kshitij Gajapure. *Drone Simulation with realistic controls made using Unity*. https://github.com/Kshitij08/Drone-Simulation. 2018.

[57] Alessio Gambi, Tri Huynh, and Gordon Fraser. "Generating effective test cases for self-driving cars from police reports". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 257–267.

[58] Alessandro Giusti et al. "A machine learning approach to visual perception of forest trails for mobile robots". In: *IEEE Robotics and Automation Letters* 1.2 (2015), pp. 661–667.

[59] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. "Automated whitebox fuzz testing." In: *Network and Distributed System Security Symposium (NDSS)*. 2008.

[60] Jun Gong et al. "Pyro: Thumb-tip gesture recognition using pyroelectric infrared sensing". In: *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. 2017, pp. 553–563.

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

[62] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. "Automated program repair". In: *Communications of the ACM* 62.12 (2019), pp. 56–65. DOI: https://doi.org/10.1145/3318162.

[63] Volker Grabe, Heinrich H Bülthoff, and Paolo Robuffo Giordano. "On-board velocity estimation and closed-loop control of a quadrotor UAV based on optical flow". In: *2012 IEEE International Conference on Robotics and Automation*. IEEE. 2012, pp. 491–497. DOI: 10.1109/ICRA.2012.6225328.

[64] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. "Automated clustering and program repair for introductory programming assignments". In: *ACM SIGPLAN Notices* 53.4 (2018), pp. 465–480. DOI: https://doi.org/10.1145/3296979.3192387.

[65] Heiko Hamann and Heinz Wörn. "A framework of space–time continuous models for algorithm design in swarm robotics". In: *Swarm Intelligence* (2008).

[66] 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.

[67] John Harwell and Maria Gini. "Improved Swarm Engineering: Aligning Intuition and Analysis". In: *arXiv preprint arXiv:2012.04144* (2020).

[68] Peter Henderson et al. "Cost adaptation for robust decentralized swarm behaviour". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2018.

[69] Fiona Higgins, Allan Tomlinson, and Keith M Martin. "Threats to the swarm: Security considerations for swarm robotics". In: *International Journal on Advances in Security* (2009).

[70] Carl Hildebrandt et al. "Feasible and stressful trajectory generation for mobile robots". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2020, pp. 349–362. DOI: https://doi.org/10.1145/3395363.3397387.

[71] Michael Hooper et al. "Securing commercial wifi-based uavs from common security attacks". In: *MILCOM 2016-2016 IEEE Military Communications Conference*. 2016.

[72] Christian Howard. *Algorithms developed to make drone swarm move together*. https://github.com/choward1491/SwarmAlgorithms. 2020.

[73] Jun S Huang et al. "An Artificial Swan Formation Using the Finsler Measure in the Dynamic Window Control". In: *Int J Swarm Evol Comput* (2020).

[74] Yu Huang et al. "Selective symbolic type-guided checkpointing and restoration for autonomous vehicle repair". In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. 2020, pp. 3–10.

[75] Ziyao Huang et al. "CoUAS: Enable Cooperation for Unmanned Aerial Systems". In: *ACM Transactions on Sensor Networks (TOSN)* 16.3 (2020), pp. 1–19.

[76] Casidhe Hutchison et al. "Robustness testing of autonomy software". In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE. 2018, pp. 276–285. DOI: 10.1145/3183519.3183534.

[77] Maximilian Hüttenrauch, Adrian Šošić, and Gerhard Neumann. "Deep reinforcement learning for swarm systems". In: *Journal of Machine Learning Research* 20.54 (2019), pp. 1–31.

[78] Maximilian Hüttenrauch, Adrian Šošić, and Gerhard Neumann. "Guided deep reinforcement learning for swarm systems". In: *arXiv preprint arXiv:1709.06011* (2017).

[79] Félix Ingrand. "Recent trends in formal validation and verification of autonomous robots software". In: *2019 Third IEEE International Conference on Robotic Computing (IRC)*. IEEE. 2019, pp. 321–328. DOI: 10.1109/IRC.2019.00059.

[80] Florida Space Institute. *EZ-RASSOR*. https://github.com/FlaSpaceInst/EZ-RASSOR. 2020.

[81] Luca Iocchi, Daniele Nardi, and Massimiliano Salerno. "Reactivity and deliberation: a survey on multi-robot systems". In: *Workshop on Balancing Reactivity and Social Deliberation in Multi-Agent Systems*. Springer. 2000, pp. 9–32. DOI: 10.1007/3-540-44568-4_2.

[82] Tom Z Jiahao, Lishuo Pan, and M Ani Hsieh. "Learning to swarm with knowledge-based neural ordinary differential equations". In: *2022 International Conference on Robotics and Automation (ICRA)*. IEEE. 2022, pp. 6912–6918.

[83] Brittany Johnson, Yuriy Brun, and Alexandra Meliou. "Causal testing: understanding defects' root causes". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 87–99.

[84] Chijung Jung et al. "SWARMFLAWFINDER: Discovering and Exploiting Logic Flaws of Swarm Algorithms". In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2022, pp. 1808–1825.

[85] Vasileios P. Kemerlis et al. "Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems". In: *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*. VEE '12. London, England, UK: ACM, 2012, pp. 121–132. ISBN: 978-1-4503-1176-2. DOI: 10.1145/2151024.2151042. URL: http://doi.acm.org/10.1145/2151024.2151042.

[86] Serge Kernbach et al. "Adaptive collective decision-making in limited robot swarms without communication". In: *The International Journal of Robotics Research* 32.1 (2013), pp. 35–55.

[87] Andrew J Kerns et al. "Unmanned aircraft capture and control via GPS spoofing". In: *Journal of Field Robotics* (2014).

[88] Dohyeong Kim et al. "Apex: Automatic programming assignment error explanation". In: *ACM SIGPLAN Notices* 51.10 (2016), pp. 311–327.

[89] Hongil Kim et al. “Touching the untouchables: Dynamic security analysis of the LTE control plane”. In: *IEEE Symposium on Security and Privacy (SP)*. 2019.

[90] Hyungsub Kim et al. “PGFUZZ: Policy-Guided Fuzzing for Robotic Vehicles”. In: ().

[91] Seulbae Kim and Taesoo Kim. “RoboFuzz: fuzzing robotic systems over robot operating system (ROS) for finding correctness bugs”. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 447–458.

[92] Seulbae Kim et al. “Drivefuzz: Discovering autonomous driving bugs through driving quality-guided fuzzing”. In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022, pp. 1753–1767.

[93] Taegyu Kim et al. “RVFUZZER: Finding input validation bugs in robotic vehicles through control-guided testing”. In: *28th USENIX Security Symposium*. 2019.

[94] kitz. *Position controller instability at yaw angles close to 180 degrees*. https://forum.bitcraze.io/viewtopic.php?t=4079. 2021.

[95] Fanxin Kong et al. “Cyber-physical system checkpointing and recovery”. In: *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE. 2018, pp. 22–31.

[96] Jinkyu Koo et al. “Pyse: Automatic worst-case test generation by reinforcement learning”. In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2019, pp. 136–147. DOI: 10.1109/ICST.2019.00023.

[97] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992. ISBN: 0262111705. DOI: 10.1007/BF00175355.

[98] C Ronald Kube and Hong Zhang. “Collective robotics: From social insects to robots”. In: *Adaptive behavior* 2.2 (1993), pp. 189–218. DOI: 10.1177/105971239300200204.

[99] D Richard Kuhn et al. “Combinatorial methods for event sequence testing”. In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 2012.

[100] Yonghwi Kwon et al. “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)*.

[101] Steven M LaValle, James J Kuffner, BR Donald, et al. “Rapidly-exploring random trees: Progress and prospects”. In: *Algorithmic and computational robotics: new directions* 5 (2001), pp. 293–308.

[102] Xuan-Bach D Le et al. “SAFFRON: Adaptive grammar-based fuzzing for worst-case analysis”. In: *ACM SIGSOFT Software Engineering Notes* 44.4 (2019), pp. 14–14. DOI: 10.1145/3364452.3364455.

[103] Claire Le Goues et al. “Genprog: A generic method for automatic software repair”. In: *Ieee transactions on software engineering* 38.1 (2011), pp. 54–72. DOI: 10.1109/TSE.2011.104.

[104] David Lewis. *Counterfactuals*. Oxford: Blackwell Publishers, 1973.

[105] J. P. Lewis. “Fast normalized cross-correlation”. In: *Proceedings of the Vision Interface*. 1995.

[106] Xin Li et al. “SWARMs ontology: A common information model for the cooperation of underwater robots”. In: *Sensors* 17.3 (2017), p. 569.

[107] Mikael Lindvall et al. “Metamorphic Model-Based Testing of Autonomous Systems”. In: *Proceedings of the 2nd International Workshop on Metamorphic Testing*. 2017.

[108] Eric Liu. *Crazyflie cannot be stable when take off, it flipped onto the ground*. https://github.com/USC-ACTLab/crazyswarm/issues/150. 2019.

[109] Yang Liu. *Swarm formation sim*. https://github.com/yangliu28/swarm_formation_sim. 2019.

[110] Yang Liu. *Swarm robot ros sim*. https://github.com/yangliu28/swarm_robot_ros_sim. 2020.

[111] Yen-Chen Liu. “Task-space control of bilateral human-swarm interaction with constant time delay”. In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2013, pp. 1663–1639. DOI: 10.1109/IROS.2013.6696572.

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

[113] Guannan Lou et al. "Testing of autonomous driving systems: where are we and where should we go?" In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 31–43.

[114] Matt Luckcuck et al. "Formal specification and verification of autonomous robotic systems: A survey". In: *ACM Computing Surveys (CSUR)* 52.5 (2019), pp. 1–41. DOI: 10.1145/3342355.

[115] Li Ma et al. "O-Flocking: Optimized Flocking Model on Autonomous Navigation for Robotic Swarm". In: *International Conference on Swarm Intelligence*. 2020.

[116] Valentin Jean Marie Manès et al. "The art, science, and engineering of fuzzing: A survey". In: *IEEE Transactions on Software Engineering* (2019).

[117] *Multi-Agent System Simulation Library*. https://github.com/TUHH-ICS/MAS-Simulation. 2022.

[118] N Harris McClamroch and Danwel Wang. "Feedback stabilization and tracking of constrained robots". In: *1987 American Control Conference*. IEEE. 1987, pp. 464–469. DOI: 10.1109/9.1220.

[119] K. N. McGuire et al. "Minimal navigation solution for a swarm of tiny flying robots to explore an unknown environment". In: (2019).

[120] *VRepRosQuadSwarm*. https://github.com/merosss/VRepRosQuadSwarm. 2016.

[121] Dejan Milutinović and Pedro Lima. "Modeling and optimal centralized control of a large-size robotic population". In: *IEEE Transactions on Robotics* 22.6 (2006), pp. 1280–1285. DOI: 10.1109/TRO.2006.882941.

[122] Robert Mitchell and Ing-Ray Chen. "Adaptive Intrusion Detection of Malicious Unmanned Air Vehicles Using Behavior Rule Specifications". In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* ().

[123] Mohammad Shameel bin Mohammad Fadilah et al. “DRAT: A Drone Attack Tool for Vulnerability Assessment”. In: *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*. New Orleans, LA, USA, 2020.

[124] Yogeswaran Mohan and SG Ponnambalam. “An extensive review of research in swarm robotics”. In: *2009 World Congress on Nature & Biologically Inspired Computing (NaBIC)*. IEEE. 2009, pp. 140–145. DOI: 10.1109/NABIC.2009.5393617.

[125] Nour Moustafa and Alireza 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*. London, United Kingdom, 2020. ISBN: 9781450381055. DOI: 10.1145/3414045.3415951.

[126] Iñaki Navarro and Fernando Matía. “An introduction to swarm robotics”. In: *International Scholarly Research Notices* 2013 (2013).

[127] Luong A Nguyen, Thomas L Harman, and Carol Fairchild. “Swarmathon: a swarm robotics experiment for future space exploration”. In: *2019 IEEE International Symposium on Measurement and Control in Robotics (ISMCR)*. IEEE. 2019, B1–3. DOI: 10.1109/ISMCR47492.2019.8955661.

[128] Changhai Nie and Hareton Leung. “A survey of combinatorial testing”. In: *ACM Computing Surveys (CSUR)* (2011).

[129] Reza Olfati-Saber. “Flocking for multi-agent dynamic systems: Algorithms and theory”. In: *IEEE Transactions on automatic control* 51.3 (2006), pp. 401–420.

[130] Ori. *DroneSimLab*. https://github.com/orig74/DroneSimLab. 2020.

[131] Sebastian Österlund et al. “Parmesan: Sanitizer-guided greybox fuzzing”. In: *29th USENIX Security Symposium*. 2020.

[132] Yash Vardhan Pant et al. “Fly-by-logic: control of multi-drone fleets with temporal logic objectives”. In: *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE. 2018, pp. 186–197.

[133] Aditya A Paranjape et al. “Robotic herding of a flock of birds using an unmanned aerial vehicle”. In: *IEEE Transactions on Robotics* (2018).

[134] Jungwon Park. *Trajectory generation and simulation for multi-agent swarm*. https://github.com/qwerty35/swarm_simulator.git. 2020.

[135] Jungwon Park et al. “Efficient multi-agent trajectory planning with feasibility guarantee using relative bernstein polynomial”. In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2020, pp. 434–440.

[136] Kunal Patel. *optimization-wolf-search-algorithm*. https://github.com/bavalia/optimization-wolf-search-algorithm. 2017.

[137] Andrea Patelli and Luca 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.

[138] David M Perry et al. “SemCluster: clustering of imperative programming assignments based on quantitative semantic features”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 860–873. DOI: 10.1145/3314221.3314629.

[139] Enrico Petritoli, Marco Cagnetti, and Fabio Leccese. “Simulation of autonomous underwater vehicles (auvs) swarm diffusion”. In: *Sensors* 20.17 (2020), p. 4950.

[140] Theofilos Petsios et al. “Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 2155–2168. DOI: 10.1145/3133956.3134073.

[141] *SWARMulator*. https://github.com/Peyje/SWARMulator. 2020.

[142] James A Preiss et al. “Crazyswarm: A large nano-quadcopter swarm”. In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2017, pp. 3299–3304. DOI: 10.1109/ICRA.2017.7989376.

[143] Ivan Pustogarov, Thomas Ristenpart, and Vitaly Shmatikov. "Using Program Analysis to Synthesize Sensor Spoofing Attacks". In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. Abu Dhabi, United Arab Emirates: Association for Computing Machinery, 2017. ISBN: 9781450349444. DOI: 10.1145/3052973.3053038.

[144] Feng Qin et al. "LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks". In: Dec. 2006, pp. 135–148. DOI: 10.1109/MICRO.2006.29.

[145] Raul Quinonez et al. "SAVIOR: Securing Autonomous Vehicles with Robust Physical Invariants". In: *29th USENIX Security Symposium*. 2020. ISBN: 978-1-939133-17-5.

[146] Ewaryst Rafajłowicz, Marek Wnuk, and Wojciech Rafajłowicz. "Local Detection Of Defects From Image Sequences." In: *International Journal of Applied Mathematics & Computer Science* (2008).

[147] Nishanth Rao. *ROS-Quadcopter-Simulation*. https://github.com/NishanthARao/ROS-Quadcopter-Simulation. 2019.

[148] Jeremias Roßler et al. "Isolating failure causes through test case generation". In: *Proceedings of the 2012 international symposium on software testing and analysis*. 2012, pp. 309–319. DOI: https://doi.org/10.1145/2338965.2336790.

[149] Dibyendu Roy et al. "Multi-robot virtual structure switching and formation changing strategy in an unknown occluded environment". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2018, pp. 4854–4861.

[150] David Rudo, Dr Zeng, et al. "Consumer UAV Cybersecurity Vulnerability Assessment Using Fuzzing Tests". In: *arXiv:2008.03621* (2020).

[151] Erol Şahin. "Swarm robotics: From sources of inspiration to domains of application". In: *International workshop on swarm robotics*. Springer. 2004, pp. 10–20.

[152] Ian Sargeant and Allan Tomlinson. "Modelling malicious entities in a robotic swarm". In: *2013 IEEE/AIAA 32nd Digital Avionics Systems Conference (DASC)*. IEEE. 2013.

[153] Charitha Saumya et al. “XSTRESSOR: Automatic generation of large-scale worst-case test inputs by inferring path conditions”. In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2019, pp. 1–12. DOI: 10.1109/ICST.2019.00011.

[154] Fabrizio Schiano and Paolo Robuffo Giordano. “Bearing rigidity maintenance for formations of quadrotor UAVs”. In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2017, pp. 1467–1474.

[155] Melanie Schranz et al. “Swarm Robotic Behaviors and Current Applications”. In: *Frontiers in Robotics and AI* 7 (2020), p. 36. DOI: https://doi.org/10.3389/frobt.2020.00036.

[156] Seong-Hun Seo et al. “Effect of spoofing on unmanned aerial vehicle using counterfeited GPS signal”. In: *Journal of Positioning, Navigation, and Timing* (2015).

[157] Seungwoo Seo, Da-Eun Ko, and Jong-Moon Chung. “Combined time bound optimization of control, communication, and data processing for FSO-based 6G UAV aerial networks”. In: *ETRI Journal* 42.5 (2020), pp. 700–711.

[158] TU Delft. *SGBA-code*. https://github.com/tudelft/SGBA_code_SR_2019. 2020.

[159] Shital Shah et al. “Airsim: High-fidelity visual and physical simulation for autonomous vehicles”. In: *Field and service robotics*. 2018.

[160] Yuju Shen et al. “Rescue: Crafting regular expression dos attacks”. In: *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2018, pp. 225–235. DOI: 10.1145/3238147.3238159.

[161] Coati Software. *Sourcetrail*. https://www.sourcetrail.com/. 2020.

[162] Yunmok Son et al. “Rocking Drones with Intentional Sound Noise on Gyroscopic Sensors”. In: *24th USENIX Security Symposium*. 2015. ISBN: 9781931971232.

[163] Dawn Song et al. “BitBlaze: A New Approach to Computer Security via Binary Analysis”. In: *Information Systems Security*. Ed. by R. Sekar and Arun K. Pujari. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–25. ISBN: 978-3-540-89862-7. DOI: 10.1007/978-3-540-89862-7_1.

[164] Enrica Soria, Fabrizio Schiano, and Dario Floreano. "SwarmLab: a Matlab Drone Swarm Simulator". In: (2020), pp. 8005–8011. DOI: 10.1109/IROS45743.2020.9340854.

[165] Siddharth Swaminathan, Mike Phillips, and Maxim Likhachev. "Planning for multi-agent teams with leader switching". In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2015, pp. 5403–5410.

[166] Swarbug. *Project Website*. https://github.com/swarmbug/src. 2020.

[167] swarm5. *ESTKALMAN: State out of bounds, resetting*. https://github.com/USC-ACTLab/crazyswarm/issues/259. 2020.

[168] swarm5. *The motor has inconsistent performance*. https://github.com/USC-ACTLab/crazyswarm/issues/289. 2021.

[169] Swarmathon. *NASA Swarmathon*. http://nasaswarmathon.com/. 2019.

[170] SwarmFlawFinder. *Project Website*. https://github.com/adswarm/src. 2021.

[171] SwarmGen. *Project Website*. https://github.com/swarmgen/src. 2023.

[172] Chris Taylor, Alex Siebold, and Cameron Nowzari. "On the effects of minimally invasive collision avoidance on an emergent behavior". In: *International Conference on Swarm Intelligence*. Springer. 2020.

[173] Haoxiang Tian et al. "Generating Critical Test Scenarios for Autonomous Driving Systems via Influential Behavior Patterns". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 2022, pp. 1–12.

[174] Christopher Steven Timperley et al. "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. DOI: 10.1109/ICST.2018.00040.

[175] Luca Della Toffola, Michael Pradel, and Thomas R Gross. "Synthesizing programs that expose performance bottlenecks". In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 2018, pp. 314–326. DOI: 10.1145/3168830.

[176] Walt Truszkowski et al. “NASA’s swarm missions: The challenge of building autonomous software”. In: *IT professional* 6.5 (2004), pp. 47–52.

[177] D. M. Tsai and C. T. Lin. “The evaluation of normalized cross correlations for defect detection”. In: *Pattern Recognition Letters* (2003).

[178] Ali E. Turgut et al. “Self-Organized Flocking with a Mobile Robot Swarm”. In: *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*. AAMAS ’08. Estoril, Portugal: International Foundation for Autonomous Agents and Multiagent Systems, 2008, pp. 39–46. ISBN: 9780981738109.

[179] Jackson State University. *Swarmathon Code of Team JSU*. https://github.com/BCLab-UNM/Swarmathon-JSU-Public. 2018.

[180] Junia Valente and Alvaro 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.

[181] Gábor Vásárhelyi et al. “Optimized flocking of autonomous drones in confined environments”. In: *Science Robotics* 3.20 (2018).

[182] N Vedachalam et al. “Autonomous underwater vehicles-challenging developments and technological maturity towards strategic swarm robotics systems”. In: *Marine Georesources & Geotechnology* 37.5 (2019), pp. 525–538.

[183] Tamas Vicsek. *Autonomous Mission Control of Drone Flocks*. Tech. rep. EOTVOS Lorand Tudomanyegetem Budapest Hungary, 2019.

[184] Anthony De Bortoli Victor Delafontaine Andrea Giordano. *A drone swarm simulator written in Matlab*. https://github.com/lis-epfl/swarmlab. 2020.

[185] Csaba Virágh et al. “Flocking algorithm for autonomous flying robots”. In: *Bioinspiration & biomimetics* 9.2 (2014), p. 025012.

[186] Di Wang and Jan Hoffmann. "Type-Guided Worst-Case Input Generation". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290326. URL: https://doi.org/10.1145/3290326.

[187] Ke Wang, Rishabh Singh, and Zhendong Su. "Search, align, and repair: data-driven feedback generation for introductory programming exercises". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2018, pp. 481–495. DOI: 10.1145/3192366.3192384.

[188] Pengfei Wang et al. "The Progress, Challenges, and Perspectives of Directed Greybox Fuzzing". In: *arXiv preprint arXiv:2005.11907* (2020).

[189] Shirley Wang et al. "Fly-Crash-Recover: A Sensor-based Reactive Framework for Online Collision Recovery of UAVs". In: *2020 Systems and Information Engineering Design Symposium (SIEDS)*. IEEE. 2020, pp. 1–6.

[190] Yan Wang et al. "A systematic review of fuzzing based on machine learning techniques". In: *PloS one* (2020).

[191] William Warke. *Crazyflie 2.1 rotating frantically and crashing at specific Yaw-Angle*. https://github.com/USC-ACTLab/crazyswarm/issues/149. 2019.

[192] Hao Wei, Jon Timmis, and Rob Alexander. "Evolving test environments to identify faults in swarm robotics algorithms". In: *IEEE Congress on Evolutionary Computation (CEC)*. 2017.

[193] Frank Willeke. *FlockModifier*. https://github.com/FlaSpaceInst/EZ-RASSOR. 2021.

[194] Sean Wilson et al. "The robotarium: Globally impactful opportunities, challenges, and lessons learned in remote-access, distributed control of multirobot systems". In: *IEEE Control Systems Magazine* 40.1 (2020), pp. 26–44.

[195] Alan FT Winfield, Christopher J Harper, and Julien Nembrini. "Towards dependable swarms and a new discipline of swarm engineering". In: *International Workshop on Swarm Robotics*. Springer. 2004.

[196] Trey Woodlief, Sebastian Elbaum, and Kevin Sullivan. "Fuzzing mobile robot environments for fast automated crash detection". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2021, pp. 5417–5423.

[197] Andrew Wright. *swarmSimRescue*. https://github.com/aywrite/swarmSimRescue. 2014.

[198] Kun Xiao et al. "Implementation of UAV Coordination Based on a Hierarchical Multi-UAV Simulation Platform". In: *arXiv preprint arXiv:2005.01125* (2020).

[199] Lingyu Yu and Victor 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.

[200] Insu Yun et al. "QSYM: A practical concolic execution engine tailored for hybrid fuzzing". In: *27th USENIX Security Symposium*. 2018.

[201] *SwarmSim*. https://github.com/yxiao1996/SwarmSim. 2020.

[202] Michal Zalewski. *American Fuzzy Lop*. http://lcamtuf.coredump.cx/afl.

[203] Andreas Zeller and Ralf Hildebrandt. "Simplifying and Isolating Failure-Inducing Input". In: *IEEE Trans. Softw. Eng.* 28.2 (Feb. 2002), pp. 183–200. ISSN: 0098-5589. DOI: 10.1109/32.988498. URL: https://doi.org/10.1109/32.988498.

[204] Ganwen Zeng and Ahmad Hemami. "An overview of robot force control". In: *Robotica* 15.5 (1997), pp. 473–482. DOI: 10.1017/S026357479700057X.

[205] Tianyao Zhang et al. "A machine learning method for vision-based unmanned aerial vehicle systems to understand unknown environments". In: *Sensors* 20.11 (2020), p. 3245.

[206] Xudong Zhang and Yan Cai. "Building Critical Testing Scenarios for Autonomous Driving from Real Accidents". In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2023, pp. 462–474.

[207] Xi Zheng et al. “On the state of the art in verification and validation in cyber physical systems”. In: *The University of Texas at Austin, The Center for Advanced Research in Software Engineering, Tech. Rep. TR-ARiSE-2014-001* (2014).

[208] Xin Zhou et al. “Ego-swarm: A fully autonomous and decentralized quadrotor swarm system in cluttered environments”. In: *2021 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2021, pp. 4101–4107.

[209] Zhiyan Zhou. *Cooperative Attack Algorithm for UAVs*. https://github.com/zzycoder/Cooperative-Attack-Algorithm-for-UAVs. 2023.

[210] Hai Zhu et al. “Distributed Multi-Robot Formation Splitting and Merging in Dynamic Environments”. In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 9080–9086.

Appendices

Appendix

A.1 Profiling the Configuration Definitions

This section explains how to profile configuration definitions, especially an approach to identify $\varnothing$ for environment configuration (Section 2.4.1).

We present how we find a value for an environment configuration variable that eliminates the causal impact of the variable. We observe that environment configuration variables have stable delta values on the lower and higher sides of mutated values. We profile these values' range (R_{avg}) by running multiple test runs. Also, we profile the average change (S_{avg}) of these variables on every tick, which essentially represents the average speed of objects, including robots and obstacles. Then, we split R_{avg} by S_{avg}, resulting in multiple groups (e.g., 100 groups if R_{avg} is 5, and S_{avg} is 0.05). We run experiments with a coordinate value from each group, observing the differences (i.e., the delta in Section 2.4.1) of the objects' coordinates from the original run. If the profiling to obtain R_{avg} and S_{avg} is insufficient, failing to find a fixed point, we multiply R_{avg} by two and repeat the experiments until it succeeds. Intuitively, this is because the lower and higher sides values (i.e., in R_{avg})of the configuration variables essentially move the object associated with the variable far away from the swarm.

Figure A.1-(b) shows computed delta values at each coordinate (x and y), and Figure A.1-(a) presents two examples of tested values (T_1 and T_2) for an obstacle. Note that O_{org} is the obstacle in the original execution which does not have an impact on the drone under test because it exists far from the drone. The silver arrow essentially represents the flight path in the original execution. We run a number of tests to cover most of the coordinates. T_1 and T_2 show two representative cases. If we mutate the obstacle's coordinate to be close to the drone T_1, it changes the drone's flight significantly, leading to a large delta value (Δ_1). On the other hand, if we mutate the coordinate to be far from the drone T_2, it does not change the drone's flight at all, leading to a "zero" delta (Δ_2).

Figure A.1-(b) presents the delta values on each coordinate. Observe that the area near the Δ_2 is all having the same delta values, which are 0, forming a flat area. The coordinate value from such a flat area is essentially a value that can eliminate the impact of the environment configuration variable. We call this value $\varnothing$. Note that different configuration variable may have different $\varnothing$ values. Hence, we repeat the above process for each variable.

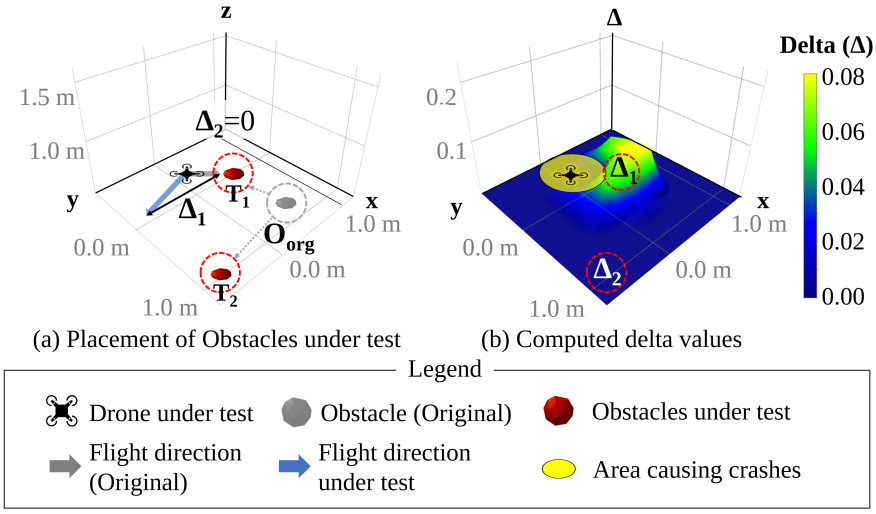


Figure A.1: Profiling configuration definitions.

A.2 Identifying the Fixed Point in Computing Spacial Variation

This section explains how to get the fixed when we repeat the process to define SVMAP.

As mentioned in Section 2.4.2, On the *ith* test set, we measure the spatial variation of the drones' poses (SV_i) from all the test runs executed at this point (i $*$ 10 tests). We repeat the process until we observe SV_{i-1} and SV_i do not differ more than 5%. In general, we reach the fixed point with 10 test sets, meaning that we run 100 tests in total.

Figure A.2-(a) is the trend of norm value of centroid of 90% area of SVMap and Figure A.2-(b) is for radius. We observe it is converged to fixed point after 9 test sets (90 tests). Note that this threshold can be differ across algorithms.

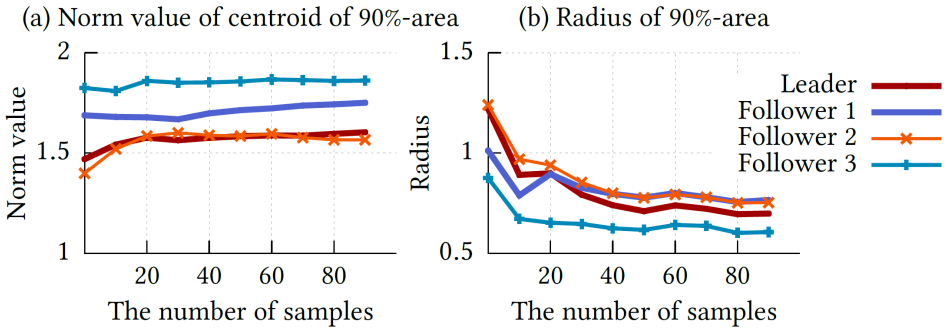


Figure A.2: Converged norm value of centroid and radius of 90% area.

A.3 Profiling the Threshold for the Time Window

This section explains how to profile the threshold for the time window. We identify when the current Dcc value is changed more than 10% than its previous tick's Dcc value (i.e., Dcc value is rapidly changing). 10% threshold is configurable and we explain how we get a time window by using this threshold. In M_k (kth mission), for m_r, we collect T_{win_i} then take an average of them. Definitions of terms are described in Section 20. In Figure A.3, we can get 6.67 tick as a time window. When we aggregate all drones (m_r) and consider more missions (M_k), this is converged into 7.6 tick.

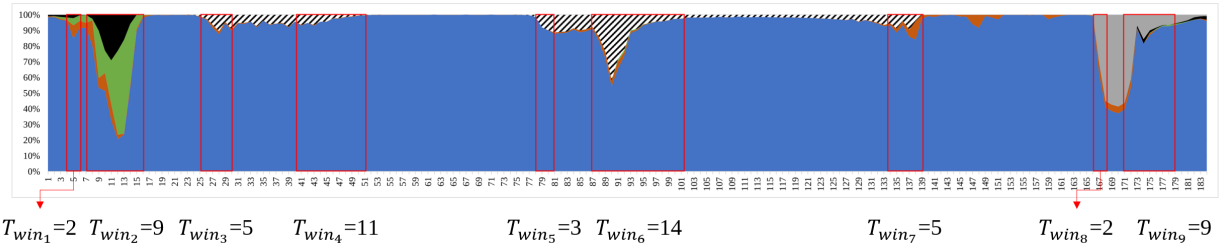


Figure A.3: Dcc value example of follower 1 (m_2).

Figure A.4 shows the partial figure for T_{win_3} and T_{win_4} in Figure A.3 and Figure A.5 shows simplified swarm's flight snapshots that correspond to Figure A.4. In ❶ of Figure A.4 and Figure A.5, with domain knowledge, we observe follower 1 flies (blue drone in above figure, before T_{win_3}) and

approaches to wall (obstacle 3) in ❷. Then, in ❸, it flies next to wall (between T_{win_3} and T_{win_4}) and tries to turn around the corner of wall (T_{win_4}) in ❹. At last, it flies away (after T_{win_4}) in ❺. So, we observe T_{win_3} and T_{win_4} are time delay between stable flight status. In this way, we can measure the T_{win}. Note that 10% threshold can be tuned for each algorithm (10% works fine for four algorithms we used in this work.)

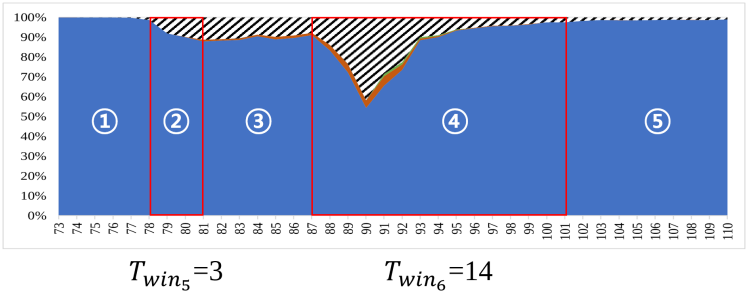


Figure A.4: Dcc value example of follower 1 (m_2).

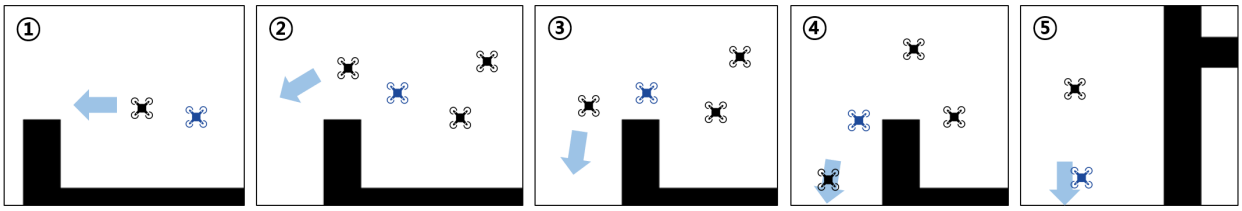


Figure A.5: Simplified swarm's flight snapshot that corresponds to Figure A.4.

A.4 Illustration of Attack Strategies

Figure A.6 illustrates the attack strategies (S_1 to S_4) in Section 3.4.1. 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 as shown in Figure A.6.

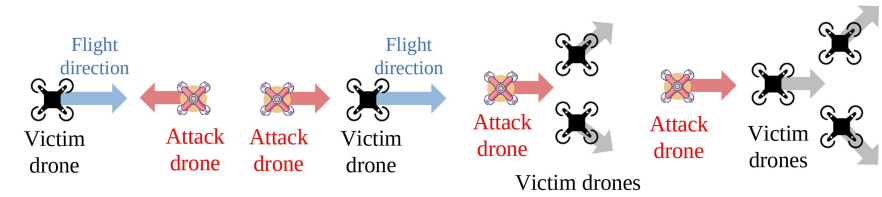


Figure A.6: Attack strategy (S).

A.5 Example Scenario for Multiple Attack Drones

Figure A.7 shows an example of multiple swarms conducting a search mission. There are two attack drones A_1 ($< P_1, S_1 >$) and A_2 ($< P_2, S_2 >$) and 11 victim drones $v_1 \sim v_{11}$. Observe that each attack drone's impact is localized: A_1 only affects a swarm with $v_1 \sim v_3$ while A_1 only impacts $v_8 \sim v_{11}$. To decide the next pose and attack strategy of A_1, v_3 is first identified since A_1 appears in the Dcc values of v_3 (i.e., A_1 directly affecting v_3). Other victim drones (v_1 and v_2) are identified because v_3 appears in other victim drones' Dcc values, indirectly affecting them. Similarly, v_9 is directly impacted by A_2, while v_8, v_{10}, and v_{11} are affected by v_9 (indirectly affected by A_2). When SwarmFlawFinder mutates $< P_1, S_1 >$, DCC values of $v_1 \sim v_3$ are used to compute NCC values. For $< P_2, S_2 >$, Dcc values of $v_8 \sim v_{11}$ are used. By doing so, even if A_1 did not lead to exercise a new behavior of the swarm $v_1 \sim v_3$, it does not affect the mutation of A_2.

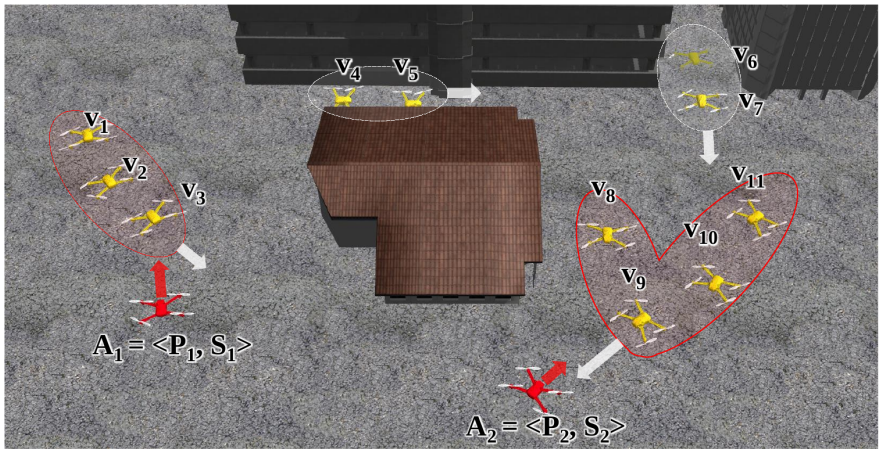


Figure A.7: Two attack drones in A3.

A.6 Additional Evaluation of the Fixes

Table A.1, Table A.2 and Table A.3 show the results of the quality of fixes for A2, A3, and A4 (A1 is presented in Section 3.5.2). Note that all the individual fixes and the integrated fixes successfully resolve the logic flaws. We do not observe any side effects for A2 (e.g., introducing new errors) as shown in Table A.1.

Table A.1: Quality of Fixes for A2

ID	Root Cause	Original (base)		fix 7 (C2-1)		fix 8 (C2-2)		fix 9 (C2-3)		fix 10 (C2-4)		int.fix	
		# of Ex.	Uniq.	# of Ex.	Uniq.	# of Ex.	Uniq.	# of Ex.	Uniq.	# of Ex.	Uniq.	# of Ex.	Uniq.
A2	**Crash***	**28**	**3**	**0**	**0**	**33**	**3**	**55**	**3**	**34**	**3**	**0**	**0**
	C2-1	28	3	0	0	33	3	55	3	34	3	0	0
	Suspended progress	**119**	**1**	**108**	**1**	**0**	**0**	**121**	**1**	**131**	**1**	**0**	**0**
	C2-2	119	1	108	1	0	0	121	1	113	1	0	0
	Slow progress	**608**	**4**	**588**	**4**	**575**	**4**	**33**	**1**	**557**	**3**	**0**	**0**
	C2-3	586	3	576	3	554	3	0	0	557	3	0	0
	C2-4	22	1	12	1	21	1	33	1	0	0	0	0
	Total:	755/990	8	696/992	5	608/972	7	209/980	5	704/981	7	0/977	0

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

We do not observe any side effects for A3 as well (e.g., introducing new errors) as shown in Table A.2.

Table A.2: Quality of Fixes for A3

ID	Root Cause	Original (base)		fix 11 (C3-1)		fix 12 (C3-2)		int.fix	
		# of Exec.	Uniq.	# of Exec.	Uniq.	# of Exec.	Uniq.	# of Exec.	Uniq.
A3	**Crash into external objects**	**47**	**2**	**33**	**1**	**9**	**1**	**0**	**0**
	C3-1	10	1	0	0	9	1	0	0
	C3-2	37	1	33	1	0	0	0	0
	Slow progress	**240**	**4**	**231**	**2**	**31**	**2**	**0**	**0**
	C3-1	23	2	0	0	31	2	0	0
	C3-2	217	2	231	2	0	0	0	0
	Total:	287/811	6	264/808	3	40/801	3	0/803	0

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

Note that all the individual fixes successfully resolve the logic flaws. However, the integrated fix fails to resolve C4-3 as shown in Table A.3. Our manual analysis suggests that this is caused by the conflict between the fixed for C4-1 and C4-3. The fix for C4-3 improves the drone's sensing sensitivity, and the fix for C4-1 makes the drone more actively avoid obstacles. When both are applied, the drone becomes extremely sensitive in avoiding obstacles, making it challenging to fly

toward a corner or narrow area. We tune the fix by reducing the sensitivity of the sensing (4 to 3). The tuned fix successfully resolves all the logic flaws without introducing additional flaws.

Table A.3: Quality of Fixes for A4

ID	Root Cause	Original (base)		fix 13 (C4-1)		fix 14 (C4-2)		fix 15 (C4-3)		int.fix	
		# of Ex.	Uniq.	# of Ex.	Uniq.	# of Ex.	Uniq.	# of Ex.	Uniq.	# of Ex.	Uniq.
A4	**Crash between Victim Drones**	**230**	**3**	**22**	**2**	**226**	**1**	**224**	**3**	**0**	**0**
	C4-1	216	1	0	0	226	1	202	1	0	0
	C4-2	14	2	22	2	0	0	22	2	0	0
	Crash into external objects	**630**	**3**	**24**	**2**	**621**	**1**	**608**	**3**	**0**	**0**
	C4-1	599	1	0	0	622	1	580	1	0	0
	C4-2	31	2	24	2	0	0	28	2	0	0
	Slow progress	**1228**	**2**	**1187**	**2**	**1233**	**2**	**0**	**0**	**134**	**2**
	C4-3	1228	2	1187	2	1233	2	0	0	134	2
	Total:	2088/2469	8	1233/2423	6	2080/2411	4	832/2481	6	134/2511	2

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

After applying each fix and the integrated fix (all fixes combined), we measure whether the patched algorithms take longer to achieve the original missions. Since the fixed swarm algorithms become more robust, it is expected to have a certain overhead. We observe 3.9%, 2.5%, 1.2%, and 1.5% average overhead for A1, A2, A3, and A4, respectively. For the integrated fix, we find that a fix with the most overhead mostly determines the overhead: 11.4%, 9.0%, 2.2%, and 4.7% average overhead for A1, A2, A3, and A4, respectively.

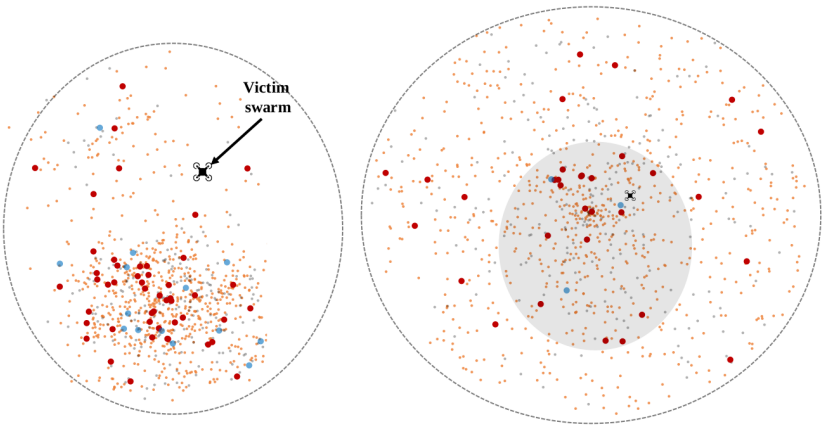
Table A.4: Normalized Overhead of Our Fixes

ID	Fix 1	Fix 2	Fix 3	Fix 4	Fix 5	Fix 6	Integrated Fix
A1	9.82%	0.84%	−0.22%	0.84%	9.69%	2.43%	11.41%
A2	3.40%	4.71%	−4.44%	6.30%	N/A	N/A	8.96%
A3	2.40%	−0.06%	N/A	N/A	N/A	N/A	2.18%
A4	2.30%	−0.43%	2.61%	N/A	N/A	N/A	4.74%

A.7 Spatial Distribution of Test Cases

Figure A.8, Figure A.9 and Figure A.10 show the results from the two versions for 24 hours of testing of A2, A3 and A4. Specifically, (a) of each figure represents the results from SwarmFlawFinder and (b) is from the random testing version. The silver round circles approximately show the size of the tested area. Each dot in the figure represents a test case. Large dots indicate they result in new

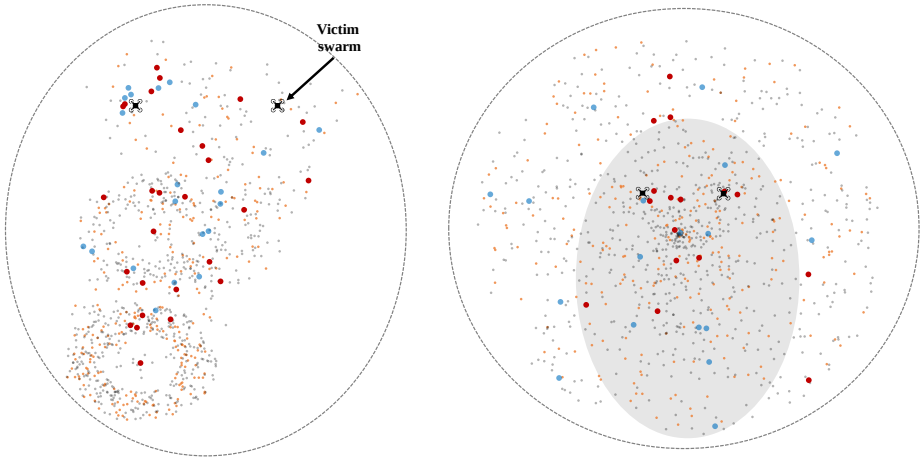
unique Dcc values, where small dots are not. Red and orange dots are the test cases that caused mission failures (i.e., discovering logic flaws). Silver and blue dots are the test cases that do not cause mission failures. The shaded areas in (b) represent the explored area by SwarmFlawFinder in (a).



(a) Visualized test cases generated for A2 by SwarmFlawFinder

(b) Visualized test cases generated for A2 by a random testing approach

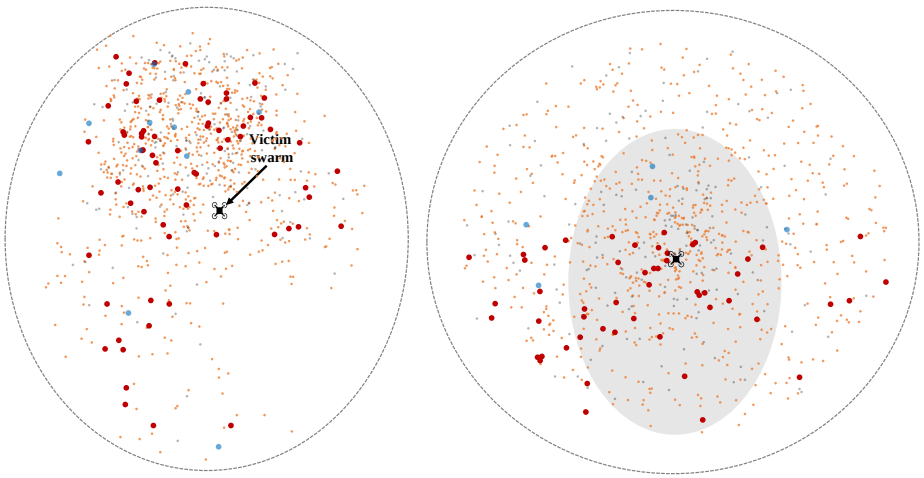
Figure A.8: Results from testing A2 with SwarmFlawFinder and Random Testing.



(a) Visualized test cases generated for A3 by SwarmFlawFinder

(b) Visualized test cases generated for A3 by a random testing approach

Figure A.9: Results from testing A3 with SwarmFlawFinder and Random Testing.



(a) Visualized test cases generated for A4 by SwarmFlawFinder

(b) Visualized test cases generated for A4 by a random testing approach

Figure A.10: Results from testing A4 with SwarmFlawFinder and Random Testing.

A.8 Activated Attack Strategies during Evaluation

Figure A.11 shows the proportions of attack strategies used during our fuzz testing evaluation in Section 3.5. Note that during our fuzz testing, we prioritize strategies that lead to new Dcc values. Hence, there can be a correlation (Not a strong correlation since there is also randomness in choosing the strategy during the test) between each strategy's effectiveness and the number of tests using the strategy.

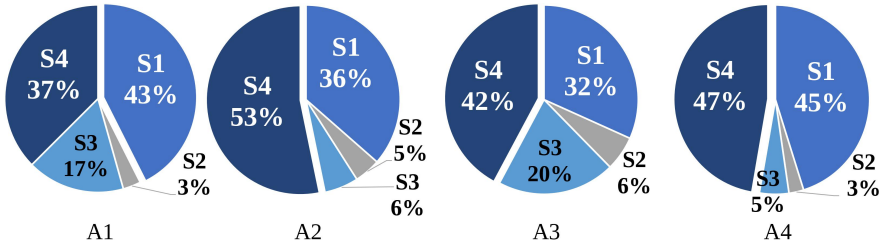


Figure A.11: Activated attack strategies on each algorithm during evaluation.

We have a few observations. First, S_1 (Pushing back) and S_4 (Herding) are the most frequently used, meaning that they might be effective on diverse swarm algorithms in general. Second, in A1 and A3, S_3 (Dividing) are frequently used (17% and 20% of all tests). This implies that the

performance of A1 and A3 depends on the coherence of the swarm. A1 needs to maintain the formation and incoherent swarms in A3 lead to many small groups of drones searching, slowing down the performance.

A.9 Details of the Number of Additional Attack Drones and Overhead

Note that our approach aims to conduct economically efficient attacks, meaning that we prefer fewer attack drones (e.g., attacks with multiple attack drones are easy but expensive). We clarify multiple attack drones scenarios as illustrated below. Specifically, the time required to conduct a single round of our experiment on an algorithm can be computed as $T_r = (N_d \times T_e) + (N_d \times T_e \times N_f)$, where N_d is the number of drones in the target (victim) swarm, T_e is the duration of a single execution of the mission, and N_f is the number of factors that can impact a victim drone's behavior. The number of factors is calculated as $N_f = (N_d - 1) + N_a + N_o$, where N_a is the number of attack drones and N_o is the number of objects in the world except for drones in target swarm and attack drones. Note that we iteratively run the experiments during our testing.

Note that to compute Dcc values, we compute delta values between the original swarm's mission and each counterfactual execution. The first part of the equation, $N_d \times T_e$, is the original swarm mission's execution. The second part of the equation, $N_d \times T_e \times N_f$, represents the counterfactual execution instances where we perturb a single factor in each execution (details in Section 3.4.2).

Table A.5: Overhead according to Additional Attack Drones

# of atk drones	Fix 1	Fix 2	Fix 3	Fix 4
+1 attack drone	8%	3%	6%	4%
+2 attack drones	14%	7%	11%	7%
+3 attack drones	21%	11%	16%	12%
+4 attack drones	28%	16%	20%	17%

As shown in the above equation of T_r, the number of attack drones is a part of N_f. In general, N additional attack drones would cause $N_d \times N$ additional execution of a mission (i.e., T_e). In

our experiment, when we add 1, 2, 3, and 4 additional attack drones, we observe 8%, 14%, 21%, and 28% overhead for Adaptive Swarm (A1), respectively. Note that the overhead with additional attack drones for other algorithms shown in Table A.5.

Observe that the overhead differs between the algorithms. There are two factors that cause the differences. First, the number of victim drones in the algorithms is different. We use 4, 8, 10, and 15 victim drones for A1, A2, A3, and A4, respectively. When the number of victim drones is small, adding attack drones causes a substantial slow down. When there are already many victim drones, adding a few does not affect the overhead. A2 is an exception in that it has fewer victim drones than A3 but has lower overhead. This is because A2 has a substantially larger codebase (e.g., it contains a large portion of the code for 3D visualization), making its vanilla execution slower than others (resulting in a large value of T_e). As a result, the impact of the number of attack drones is reduced.

A.10 Trend of Complexity Score for A2, A3, and A4

We measure the changes in complexity score while mutating. Figure A.12, Figure A.13 and Figure A.14 show the changes of complexity score of A2, A3, and A4. We also observe that the fixed points depend on the space in each mission for the drone's possible path, as explained in Section 4.4.4 with A1's case.

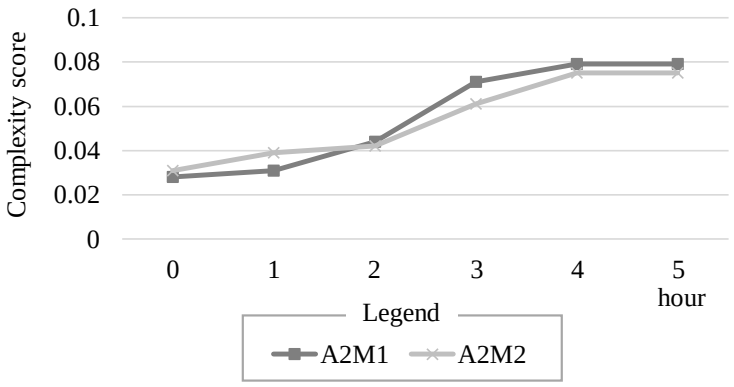


Figure A.12: The complexity score of missions in A2.

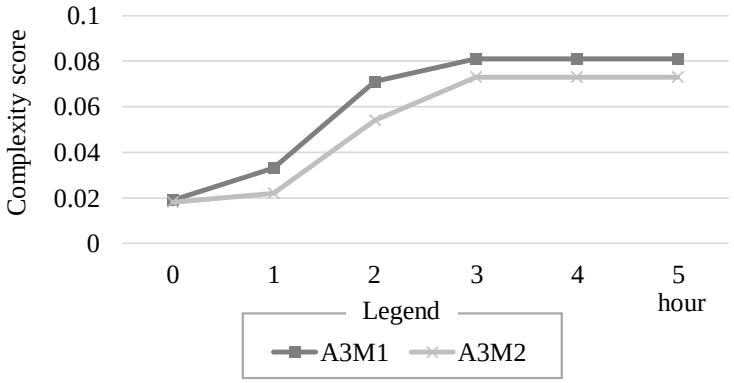


Figure A.13: The complexity score of missions in A3.

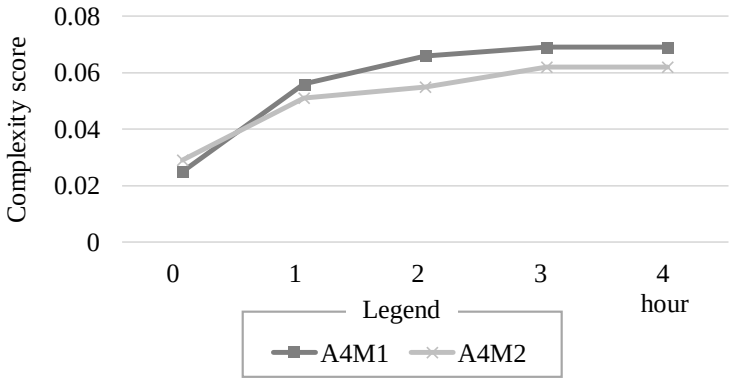


Figure A.14: The complexity score of missions in A4.