Fuzzing Mobile Robot Environments for Fast and Automated Crash Detection

Trey Woodlief[1], Sebastian Elbaum[2], and Kevin Sullivan[3]

***Abstract*— Testing mobile robots is difficult and expensive, which leads to many faults going undetected. In this work we explore whether fuzzing, an automated test input generation technique popular for software, can assist in quickly finding failure inducing inputs in mobile robots. We developed a simple fuzzing adaptation, BASE-FUZZ, and one specialized for fuzzing mobile robots, PHYS-FUZZ. PHYS-FUZZ is unique in that it accounts for physical attributes such as the robot dimensions, estimated trajectories, and time-to-impact measures to guide the test input generation process. The results of PHYS-FUZZ evaluation suggest that it has the potential to speedup the discovery of input scenarios that reveal failures, finding 56.5% more than random input selection and 7.0% more than BASE-FUZZ during 7 days of testing.**

I. INTRODUCTION

Testing mobile robots is difficult and expensive for at least three reasons. First, the input space is large and complex, including the robot and the environment state, making it challenging to cover it extensively through testing [23]. Second, although the test execution can be automated through simulators [24], and the input space codified [4], the generation of test scenarios remains mostly a manual process [1], [16]. Third, developing oracles to judge correctness requires handling large ranges of possible behaviors, noise in measuring system response, and non-determinism in robot performance [1]. As a result, system test suites for robots, from the PX4 [16] to the TurtleBot [18], [19] to the Care-O-bot [3], [10], tend to consist of a small number of hand-selected inputs with a human acting as an oracle or with carefully crafted oracles, or just scenarios on which to add a goal and an oracle to form a test, potentially missing many corner cases [1], [9].

In this work we explore whether fuzzing, a software technique to quickly expose system failures through guided input generation, can address such challenges. Fuzzing was defined decades ago by Miller et al., who utilized randomly generated input strings to exercise unix utilities and uncover failures [17]. Fuzzing is appealing because of its low overhead for adoption due to using high-level oracles that require no additional effort on the part of the developers, and automatic input generation that learns from its previous findings.

Modern techniques and supporting tool sets [6], [7], [25] have improved fuzzing cost-effectiveness by guiding the random input generation with sophisticated feedback mechanisms. Such feedback mechanisms utilize, for example, the lack of coverage of code constructs to favor inputs that may traverse uncovered predicate branches or predicates that resulted in a failure in previous tests. The success of fuzzing in the software arena is undeniable [13], as it is now a staple in software validation processes.

Applying fuzzing to mobile robots has the potential to address some of the challenges enumerated earlier by more effectively sampling the input scenarios of mobile robots while using simple high-level oracles to automatically judge incorrect behavior. Yet, to be valuable to the validation of mobile robots, we believe that fuzzing should incorporate two fundamental changes. First, it must shift from feedback based on code constructs, to feedback that captures the distinct physical features of the robot and the environment to guide scenario generation. Second, it should shift from oracles that consider binary pass/fail test outcomes, to ones that judges outcomes based on more continuous *hazardousness* measurements associated, for example, with nearness to collision states.

We introduce an approach for the fuzzing of mobile robots, PHYS-FUZZ, that takes such a leap. By encoding physical features in a manner that respects their real-world interpretation and using continuous metrics of robot performance, PHYS-FUZZ can more quickly guide fuzzing to yield system failures. Our contributions are:

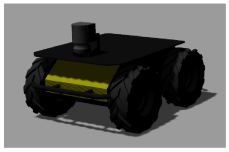
- Defining an approach, PHYS-FUZZ, for fuzzing mobile robots, which integrates traditional notions of software fuzzing with the physical attributes and hazards of mobile robots and their environments.
- Developing a family of techniques that implement PHYS-FUZZ, accounting for physical attributes such as dimensions of the robot, estimated trajectories, and time-to-impact measures and using automatically generated oracles based on the hazardousness of a test.
- Assessing the potential of PHYS-FUZZ when compared with random and traditional fuzzing techniques to accelerate the detection of physical robot collisions, in a mobile robot built on ROS and executed under Gazebo.

II. MOTIVATING EXAMPLE

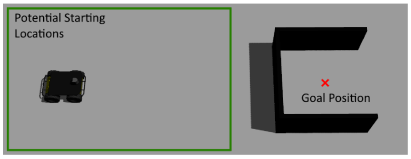
The Husky unmanned ground vehicle is a robot research and development platform produced by Clearpath Robotics that is popular for its extensibility and off-the-shelf configurations with supporting open-source code [2], [22]. One of the default hardware configurations, pictured in Figure

[1]University of Virginia, USA, *adw8dm@virginia.edu*
[2]University of Virginia, USA, *selbaum@virginia.edu*
[3]University of Virginia, USA, *sullivan@virginia.edu*

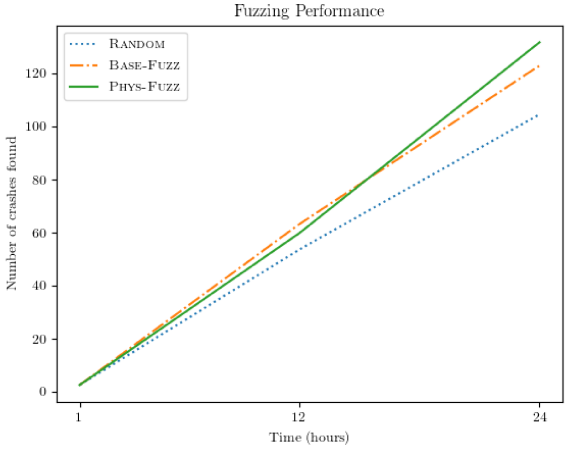
This work was funded in part by NSF Awards #1853374 and #1909414. Trey Woodlief was supported in part by a University of Virginia SEAS Fellowship



(a) Husky Robot



(b) Simple go-to-goal scenario example



(c) Collisions detected by techniques

Fig. 1: Motivating Example. Husky Robot in simple scenario navigating to goal within a garage.

1a uses a LIDAR for sensing, allowing out-of-the-box navigation capabilities. Let's now set this Husky on a simple mission. The Husky has been tasked with navigating to a location that is within a 2 meter square garage walled off on 3 sides. The robot can start at any position within a pre-defined 4 meter by 10 meter region facing 0°, and must navigate to the goal without crashing into the garage.

To emphasize the testing challenges even in the most basic setting, in this motivating example, testing has been simplified to the selection of concrete values for just the starting pose of the robot, as demarcated in the green area in Figure 1b. An approach to select such values may simply select random poses within the starting area. This naive approach, however, may generate many similar tests that render limited value in exploring the potential behaviors of the Husky.

A slightly more sophisticated approach may partition the space in some way, perhaps using the method of Rapidly-exploring Random Trees (RRT) [12] or by having the developer specify reasonable areas of say 20cm by 20cm. Such partitions may still render many potential tests, 1000 if we follow the 20cm by 20cm partitioning. Given that these tests take on average 60 seconds to run in a Gazebo simulation, running the whole suite would take almost 17 hours. That cost is compounded by the fact that the robot may exhibit non-deterministic behavior, and thus each test should be run multiple times to assess its failure probability. For our study, those runs caused the complete test suite to take over 80 hours. It is easy to imagine how for even slightly more complex scenarios such an approach becomes infeasible.

The problem we are exploring, then, is how we can more quickly identify what test scenarios are worth generating and executing early in order to accelerate the detection of failing robot behaviors that result in crashes.

Figure 1c shows the number of crashes found by three techniques when testing for up to 24 hours. The RANDOM technique explores the scenario space by sampling using a uniform distribution. The BASE-FUZZ technique is a direct adaption of standard fuzzing for robots where the selection of the next test to execute is biased towards tests that are most similar to tests that have exhibited crashes in the past. The underlying notion of test similarity is based, for example, on the initial pose of the robot relative to the obstacle. Finally, the PHYS-FUZZ technique considers the physical meaning of the testing scenarios and is guided by a *hazardousness score* from the robot's performance in the test. That is, PHYS-FUZZ biases selection towards tests with similar features such as the expected trajectory, the sensed environment, and also considers a continuous distinction of test outputs based on time-to-impact (instead of the binary non-crash/crash).

In the first several hours, while the fuzzing techniques are still randomly exploring the space, all three techniques perform comparatively. And for this particular scenario, RANDOM performs rather well uncovering many failures. However, the difference between techniques becomes evident as more data helps to refine the models to bias the test to generate and execute. At the 24 hour mark, PHYS-FUZZ has uncovered 26.0% more collisions than RANDOM, and 6.5% more crashes than BASE-FUZZ. Even for a simple scenario space with limited choices, by taking advantage of fuzzing and the physical aspects of robots, PHYS-FUZZ is able to uncover more crashes faster than both baseline techniques.

III. APPROACH

The goal of PHYS-FUZZ is to more quickly uncover scenarios in which a mobile robot exhibits failing behaviors leading to crashes. This process is divided into three phases: run, learn, and select. Fuzzing gathers information from running tests, learns from the results, and finally selects new tests designed to yield failures and further aid in the learning process.

A. Algorithmic Description

Algorithm 1 is the core for fuzzing a mobile robot in the physical world. It takes as input a test scenario space and a robot specification. A scenario space consists of a tuple $(\mathcal{S}, \mathcal{O}, \mathcal{G})$. $\mathcal{S}$ describes the set of starting configurations for the robot, $\mathcal{O}$ is a set describing the possible types, quantities, and poses of all obstacles, and $\mathcal{G}$ describes the set of objectives for the robot, e.g. the set of possible goal poses. A concrete scenario is described by the tuple (s, o, g), where s selects a starting pose in the world frame; o is a set of obstacles including their shapes and poses in the world frame; and g is a goal action. The algorithm also takes as input information about the robot under test including its shape and kinematic model.

Algorithm 1 Fuzzing for Mobile Robots

procedure FUZZROBOTS($scenarioSpace, robot, initialScenarioCount, timeout$)
  $scenarios[] =$ GENERATE($scenarioSpace, robot, [], initialScenarioCount$)
  while $\neg$TIMEOUTELAPSED($timeout$) **do**
    $learnedRelations =$ LEARN($scenarios[]$)
    $scenario =$ SELECT($scenarios[], learnedRelations$)
    $scenario.data =$ RUN($scenario$)
    $scenario.hazardous =$ COMPUTEHAZARDOUSSCORE($scenario.data, robot$)
    $scenarios[] =$ GENERATE($scenarioSpace, robot, scenarios[], initialScenarioCount$)
  end while
end procedure

Algorithm 2 Scenario Generation

procedure GENERATE($scenarioSpace, robot, scenarios, initialScenarioCount$)
  if NEEDMORETOCONSIDER($scenarios[]$) **then**
    $moreScenarios[].world =$ GENSCENARIOS($scenarioSpace, initialScenarioCount$)
    $moreScenarios[].staticFeatures =$ COMPUTESTATIC($scenarioSpace, moreScenarios[]$)
    $moreScenarios[].physicalFeatures =$ COMPUTEPHYSICALSTATIC($scenarioSpace, moreScenarios[], robot$)
    $scenarios.append(moreScenarios)$
  end if
  return $scenarios[]$
end procedure

The algorithm begins by using GENERATE to initialize a set of $initialScenarioCount$ test scenarios to consider for execution. These scenarios are sampled uniformly at random from the scenario space and enable the initial computing of static and physical features. For each of these scenarios, two sets of features are computed without executing the scenario. We call these static features. COMPUTESTATIC calculates a set of static features that encode the scenario tuple described earlier. Next, COMPUTEPHYSICALSTATIC calculates a set of features over physical aspects of the scenario. We describe their differences in more detail in the next section. The main loop then executes tests for a specified period of time as defined by $timeout$. The algorithm uses LEARN to process the data from previous executions to learn how to build associations between the static features and the hazardousness score to guide test selection. SELECT uses the learned relations to choose a test scenario to run based on its predicted likelihood of revealing a hazard. The scenario can then be executed either in simulation or in the real world as represented by RUN. The test execution is then analyzed and COMPUTEHAZARDOUSSCORE evaluates the robot's performance. This score is used in the next iteration to guide fuzzing toward more hazardous scenarios. Methods for implementing COMPUTEHAZARDOUSSCORE are described in section IV-B. GENERATE then uses a heuristic to determine if the algorithm has enough unexecuted scenarios remaining to consider, and, if not, repeats the scenario generation and feature computation steps.

B. Algorithm Instantiations

The simplicity of our framework belies its flexibility to implement many potential fuzzing techniques. We explore several instantiations of Algorithm 1 of increasing complexity.

RANDOM explores the scenario space by uniformly sampling from $(\mathcal{S}, \mathcal{O}, \mathcal{G})$. All of the calls to the COMPUTESTATIC, COMPUTEPHYSICALSTATIC, LEARN, and COMPUTEHAZARDOUSSCORE can be omitted as the SELECT function does not consider any scenario information when choosing the next scenario to test.

BASE-FUZZ adapts standard fuzzing techniques to robotics. As we shall see, automating that in itself can provide important gains over RANDOM. This approach explores the scenario space using only static features. COMPUTESTATIC generates static features derived automatically from the scenario tuple. This technique does not associate any physical meaning to those features. As such, it could be applied to any software running on any system. COMPUTEHAZARDOUSSCORE under this technique evaluates whether the robot passed or failed the test and returns a binary value accordingly. The implementation of LEARN is discussed in section IV-B. COMPUTEPHYSICALSTATIC is omitted.

PHYS-FUZZ explores the scenario space using all available static and physical features, and shifts to a continuous hazardousness score. This score integrates non-physical data computed by COMPUTESTATIC and physical information about the environment computed by COMPUTEPHYSICALSTATIC. Physical features capture the environment explicitly. This is a novel addition beyond standard fuzzing because the encoding requires recognition and special treatment of different physical quantities. For example, consider that 0° and 350° are 10° apart, not 340° as a direct mapping would imply. Angles do not map well to a single value, and instead the corresponding physical feature $(cos(\theta), sin(\theta))$

allows for the comparison of any two angles while respecting their physical meaning. The physical features also include information about how the robot interacts with the scenario, for example, the shortest path for the robot to reach the goal.

COMPUTEHAZARDOUSSCORE under PHYS-FUZZ captures information about how the robot responded to the physical environment such as how close the robot came to colliding with an obstacle. Instead of rating a test as only pass or fail, a continuous metric describes how close the robot was to failing during a successful execution. This technique explicitly encodes the physical nature of the tests and LEARN can consider varying degrees of success. The implementation of LEARN is discussed in section IV-B.

IV. IMPLEMENTATION

The BASE-FUZZ and PHYS-FUZZ techniques described above use machine learning regression models implemented using Scikit-learn [20] to guide fuzzing. These models reside in the LEARN method. In both techniques, each scenario has a feature vector, x, and a learning objective, y (described in more detail in the next sections). The model is then trained to predict the learning objective y' of an unexecuted scenario using its feature vector x'. The feature vector is formed from the static and physical features produced by the technique. This underscores the importance of these features being static and precomputable as they are used as the input to predict the target score before executing the scenario.

For example, consider an input space that has one obstacle, placed either at 5 or 10 meters away from the robot and at either 0, 45, or 315 degrees. Now let's suppose a concrete testing scenario places the goal at 5 meters and an orientation of 45° and results in a crash. BASE-FUZZ would encode this as $x = (1, 0, 0, 1, 0), y = 0$. PHYS-FUZZ would encode the same scenario as $x = (1, 0, 0, 1, 0, 5m/10m, cos(45°), sin(45°)) = (1, 0, 0, 1, 0, 0.5, \sqrt{2}/2, \sqrt{2}/2), y = 0$. Now, a second concrete testing scenario places the goal at 10 meters at an angle of 315° and results in the robot having a near miss with an obstacle but reaching the goal anyway. BASE-FUZZ would encode this as $x = (0, 1, 0, 0, 1), y = 1$. PHYS-FUZZ would encode the same scenario as $x = (0, 1, 0, 0, 1, 10m/10m, cos(315°), sin(315°)) = (0, 1, 0, 0, 1, 1, \sqrt{2}/2, -\sqrt{2}/2), y = 0.1$. This value of $y = 0.1$ is slightly above 0, representing an execution that came close to crashing but passed. Note that as the angle changed from 45° to 315°, PHYS-FUZZ was able to identify the physical symmetry through the use of cos and sin.

A. Feature Vectors

Our fuzzing techniques compute a single feature vector for each scenario. Static features are encoded using a one-hot encoding since the values represented by such features lack physical context. Physical features about the environment represent concrete values and the difference between two values gives information about their relative pose in the real world. Thus, physical features are scaled to be in the range 0-1 based on the maximum possible value for the feature as defined by the scenario space and then encoded directly, preserving their relative distance. Our fuzzer also uses the robot's kinematic model to compute the shortest path for the robot. This is included in the feature set by adding a single feature of the sum of the absolute value of the total angles that the robot must turn to complete the path. As illustrated earlier, the feature vector for BASE-FUZZ is exactly the one-hot encoding of the static features. For PHYS-FUZZ, the feature vector is the concatenation of the one-hot encoded static features and the physical features.

B. Learning

The goal of both fuzzing techniques is to generate scenarios that lead to failures as fast as possible, guided by the learning objective given by COMPUTEHAZARDOUSSCORE. However, each technique implements COMPUTEHAZARDOUSSCORE in a different manner. With BASE-FUZZ, the only guiding information is a binary indicator for pass or fail.

For PHYS-FUZZ, the hazardousness score is calculated on a continuous scale representing the level of hazard the robot encountered. The robot sensor data is used to calculate the time to impact at all times during the execution. Crashes have a time to impact of 0 while low times indicate a near miss and high times indicate the robot was not in danger of colliding. To prevent infinite values from biasing a scenario, any values greater than 10 seconds were set to 10. To condense this measure to a single number per scenario, our technique finds the one second window in which the average time to impact is lowest and uses that average as the candidate value, v in seconds. Time to impact proportionally encodes what we think of as the *hazardousness* of an execution. To discriminate between a crash and near-miss we also apply the function $h(v) = T(1 - e^{-2v/T})$ where T is a scaling factor, to spread out the range near zero. T was set to 10 so that the range of the non-linear function matched the original values.

The models for each technique are then trained on their respective hazardousness scores. The unexecuted scenario that is predicted to be the most hazardous is then selected. By using this continuous metric, PHYS-FUZZ has more information and can begin guiding fuzzing sooner. BASE-FUZZ must find a crash before it has any actionable data. By contrast, even before a crash is found PHYS-FUZZ can differentiate scenarios and guide toward hazardous scenarios.

In terms of learning models, we find that many machine learning regression models can be applied to this problem. In practice, a K-nearest-neighbors regression model was found to most reliably provide the best guidance for both BASE-FUZZ and PHYS-FUZZ. The optimal number of neighbors to consider varies based on the total number of scenarios that are considered. Empirically we found a value of $\approx 0.01 * initialScenarioCount$ to be suitable.

C. Running Tests

The tests were run in simulation using Gazebo 9.0.0 [5], [11] running inside of a Docker container using ROS [21].

We developed an automated framework in Python for generating and executing tests in Gazebo. With a small overhead to handle a specific scenario space, the system produces a Gazebo environment and automatically executes the test for the robot with additional ROS nodes added to send the goal to the robot, monitor for when a collision has occurred, and log data used for calculating the hazardousness score.

Running tests in simulation allows for multiple tests to be executed at once given sufficient hardware availability. Instead of selecting the single scenario that is scored most hazardous, the model can select a batch of the most hazardous scenarios to be run in parallel since the feedback loop does not need to run after every scenario. This optimization was applied during the study, as discussed in section V-A.

Robots operate in an uncertain environment and often exhibit non-deterministic behavior. As such, many tests are flaky [14] and may only fail some portion of the time [1], [8]. To address this, the selected scenario should be run multiple times. After all executions are complete, the results and data of each of the executions are used to compute separate learning objectives per execution. The machine learning model used must be robust to handle having the same feature vector appear multiple times with different learning objectives.

V. Study

Our study aims to answer the following research questions:

RQ1. What is PHYS-FUZZ's cost-effectiveness when compared to RANDOM and BASE-FUZZ? We will measure cost-effectiveness in terms of failures found over units of time.

RQ2. Can PHYS-FUZZ find failures in meaningfully different contexts? How does it compare with RANDOM and BASE-FUZZ?

A. Study Design

The study was performed using the Husky robot [22] and the provided ROS Melodic navigation source code [2]. The Husky is a 990mm by 670mm robot designed with 4 independent wheels allowing it to rotate in place. The Husky was configured with a SICK LMS100 LIDAR scanner with 360° view and 10m range for sensing. In all scenarios the robot was provided with a world map for navigation. The Husky uses the ROS move_base [15] navigation framework which takes a goal location as input and attempts to navigate to that location. If navigation becomes infeasible, the system will return an abort code, otherwise it will return success.

The scenario space is described by the tuple $(\mathcal{S}, \mathcal{O}, \mathcal{G})$. $\mathcal{S}$ consisted of a single starting pose at the origin oriented toward 0°. The obstacle space $\mathcal{O}$ consists of exactly two U shaped obstacles. One obstacle surrounds the starting location facing 0° and another is placed 10 meters away at one of 8 locations at 45° increments. Additionally, the second goal can vary its orientation in 45° increments starting at 0°. Each obstacle can independently vary the width and depth of the opening between 2, 3, and 4 meters. $\mathcal{G}$ consisted of one goal, to navigate from the starting location to the center of the second obstacle without colliding. Figure 2 shows an example scenario from the space. There are 8 locations and 8 orientations for the second obstacle and 3 possible sizes for each of the width and height of both obstacles. This leads to 8*8*3*3*3*3=5184 possible concrete scenarios generated from this space, all of which are feasible for the Husky.

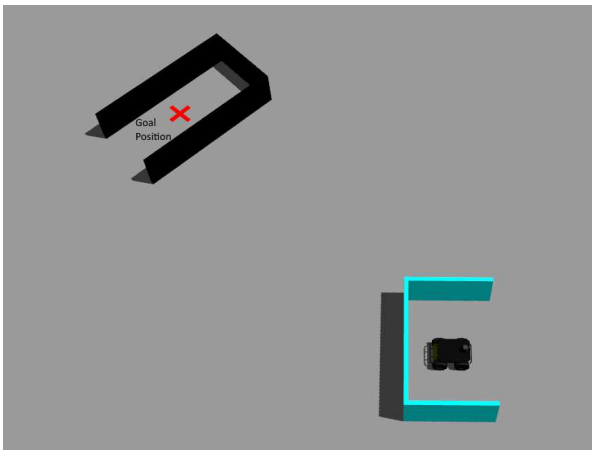


Fig. 2: Sample study generated scenario

The techniques then operated as in Algorithm 1 to generate scenarios, learn, and select which to run[1], except calls to RUN returned the data cached from disk for the scenario. This allows for a comparison of the techniques independent of the non-determinism introduced by the flakiness of the tests. The techniques are then evaluated in their ability to find scenarios leading to crashes over time. With 5184 scenarios each with 5 executions, there were 25920 total executions. Of these, 21497 (82.9%) were successful, 3061 (11.8%) resulted in the robot aborting the mission, 1340 (5.17%) resulted in collisions, and 22 did not finish due to either simulator error or reaching the 10 minute timeout. Aborts fall into two categories: the robot has gotten itself into a hazardous situation and aborted for robot safety or the robot encounters an internal error and must stop. In the former case, the time to impact still provides valuable information as the robot has itself identified a hazardous situation. However for the latter case, the time to impact may not provide useful information since the internal error may not be related to the environment. To handle this, any executions that resulted in an abort and never had a finite time to impact were ignored during learning. To take advantage of the system's ability to run multiple containers in parallel, the optimization noted in section IV-C was applied. Each call to SELECT produced a list of 50 scenarios to execute.

B. RQ1 Results: Cost-effectiveness

Figure 3 shows the performance of the three techniques during the first 7 CPU-days of execution. Because the initial learning periods randomly select scenarios to execute, the

[1]Tests were run on a pop-os 18.04 desktop with an Intel Xeon Silver 4216, 128GB of RAM, and an NVIDIA TITAN RTX GPU with 24GB of VRAM. Each scenario's 5 executions were run in an Ubuntu 18.04 Docker container. Up to 12 containers were run in parallel.

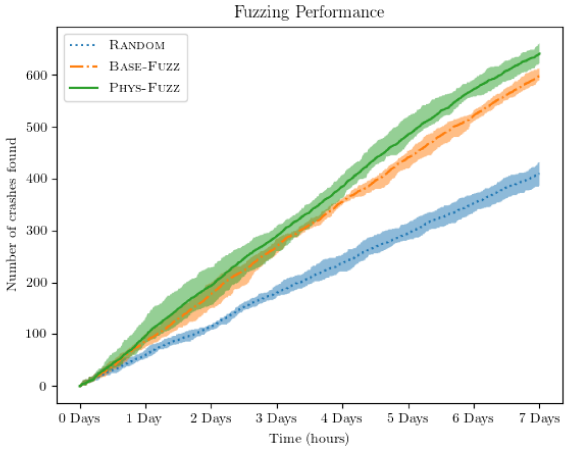


Fig. 3: Crashes found during 7 days of testing

random seed used can influence the performance of the technique. All techniques were executed using 5 random seeds and the lines in Figure 3 show the average performance across the seeds with shading indicating the range of values.

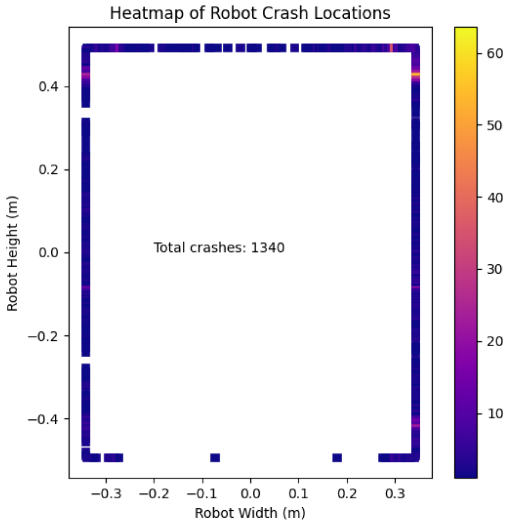
After 7 days, RANDOM was able to find on average 409 tests that resulted in crashes. BASE-FUZZ was able to find 598 crashes which is a 46.2% increase over RANDOM. Meanwhile PHYS-FUZZ was able to find 640 collisions, a 56.5% increase over RANDOM and a 7.0% increase over BASE-FUZZ. The maximum difference between PHYS-FUZZ and RANDOM was 69.0% at 50 hours, and the maximum difference between PHYS-FUZZ and BASE-FUZZ was 11.1% at 106 hours. Further, after the 95 hour mark, PHYS-FUZZ outperforms BASE-FUZZ fuzzing across all random seeds.

From a different perspective, consider a team of engineers waiting for faults to be found in the system in order to be fixed. PHYS-FUZZ would deliver in 4.25 days the same number of crashes that would take RANDOM 7 days.

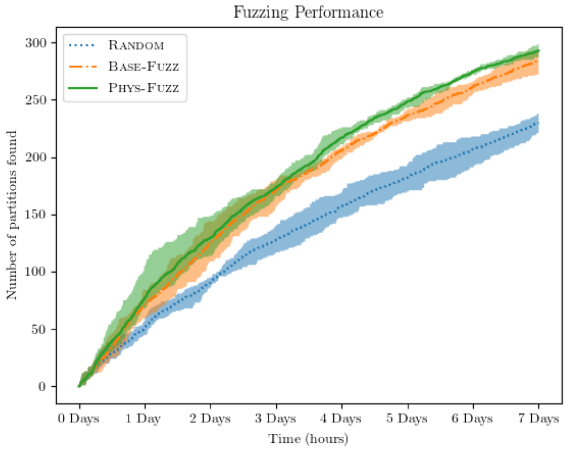
C. RQ2 Results: Discovering Different Failures

To evaluate the ability of the techniques to find failures in meaningfully different contexts, the crashes were separated based on what portion of the robot was involved with the crash. The perimeter of the Husky was divided into 720 sectors of equal angle measured from the center of the vehicle. Each crash was then categorized based on the point of impact.

The 1340 total crashes observed through all generated tests covered 435 distinct impact partitions. The partitions are visualized in Figure 4a. Figure 4b shows the number of partitions found over time. On average, RANDOM finds 214 crash partitions, compared to the 247 from BASE-FUZZ and 251 from PHYS-FUZZ. While both fuzzing techniques outperform RANDOM by at least 15% with a slight edge to PHYS-FUZZ, BASE-FUZZ and PHYS-FUZZ perform similarly in terms of partitions found even though data from RQ1 established PHYS-FUZZ has found more crashes overall. This is not surprising as neither technique was tailored to uncover



(a) Visualization of crash partitions



(b) Different crash classes found during execution

distinct crashes but rather more crashes. Software fuzzers can be tuned for finding many failures (depth) or many distinct failures (breadth), which remains to be explored in the context of mobile robot fuzzing.

VI. CONCLUSION

The results of our exploration and evaluation of PHYS-FUZZ suggest that it has the potential to meaningfully speed up the discovery of fault-revealing inputs for mobile robots. The findings also set several directions for future work. First, we would like to explore more complex scenarios with richer physical features and also more sophisticated systems that have associated hazard models. Second, we would like to investigate how to leverage more sophisticated fuzzing mechanisms that consider both the limited availability of testing resources and the need to maximize the number of faults exposed, and that can be tuned for breadth versus depth.

REFERENCES

[1] A. Afzal, C. L. Goues, M. Hilton, and C. S. Timperley. A study on challenges of testing robotic systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 96–107, 2020.

[2] Clearpath Robotics et al. *Husky GitHub Repository*, 2020 (accessed October 27, 2020). https://github.com/husky/husky/tree/melodic-devel.

[3] Fraunhofer IPA et al. *Care-O-bot GitHub Repository*, 2020 (accessed October 30, 2020). https://github.com/ipa320/care-o-bot.

[4] Daniel J Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L Sangiovanni-Vincentelli, and Sanjit A Seshia. Scenic: a language for scenario specification and scene generation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 63–78, 2019.

[5] Gazebo. Robot simulation made easy. http://gazebosim.org, 2014. 2020 (accessed October 28, 2020).

[6] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20–27, 2012.

[7] Gustavo Grieco, Martín Ceresa, and Pablo Buiras. Quickfuzz: An automatic random fuzzer for common file formats. *ACM SIGPLAN Notices*, 51(12):13–20, 2016.

[8] C. Hutchison, M. Zizyte, P. E. Lanigan, D. Guttendorf, M. Wagner, C. Le Goues, and P. Koopman. Robustness testing of autonomy software. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 276–285, 2018.

[9] Eliahu Khalastchi and Meir Kalech. On fault detection and diagnosis in robotic systems. *ACM Computing Surveys (CSUR)*, 51(1):1–24, 2018.

[10] Ralf Kittmann, Tim Fröhlich, Johannes Schäfer, Ulrich Reiser, Florian Weißhardt, and Andreas Haug. Let me introduce myself: I am care-o-bot 4, a gentleman robot. In Sarah Diefenbach, Niels Henze, and Martin Pielot, editors, *Mensch und Computer 2015 – Proceedings*, pages 223–232, Berlin, 2015. De Gruyter Oldenbourg.

[11] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2149–2154, Sendai, Japan, Sep 2004.

[12] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[13] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[14] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 643–653, 2014.

[15] Eitan Marder-Eppstein. *ROS move_base*, 2020 (accessed October 31, 2020). http://wiki.ros.org/move_base.

[16] Lorenz Meier, Dominik Honegger, and Marc Pollefeys et al. *PX4-Autopilot GitHub Repository*, 2020 (accessed October 29, 2020). https://github.com/PX4/PX4-Autopilot.

[17] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[18] Inc. et al Open Source Robotics Foundation. *Turtlebot GitHub Repository*, 2020 (accessed October 30, 2020). https://github.com/turtlebot/turtlebot/tree/melodic.

[19] Inc. et al Open Source Robotics Foundation. *Turtlebot Simulator GitHub Repository*, 2020 (accessed October 30, 2020). https://github.com/turtlebot/turtlebot_simulator/tree/melodic.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[21] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[22] Clearpath Robotics. *Husky Unmanned Ground Vehicle*, 2020 (accessed October 30, 2020). https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/.

[23] Zaid Tahir and Rob Alexander. Coverage based testing for v&v and safety assurance of self-driving autonomous vehicle: A systematic literature review. In *The Second IEEE International Conference On Artificial Intelligence Testing*. York, 2020.

[24] Christopher Steven Timperley, Afsoon Afzal, Deborah S Katz, Jam Marcos Hernandez, and Claire Le Goues. Crashing simulated planes is cheap: Can simulation detect robotics bugs early? In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 331–342. IEEE, 2018.

[25] Michał Zalewski. *American Fuzzy Lop*, 2013 (accessed October 27, 2020). https://lcamtuf.coredump.cx/afl/.