DiscoFuzzer: Discontinuity-based Vulnerability Detector for Robotic Systems

***Abstract*—Robotic systems continue their diffusion accomplishing physical tasks on our behalves. However, their safety-critical nature implies the cruciality of testing their robustness. In this paper, we propose a novel fuzzing methodology that exploits the continuity of the physical world to automatically explore the input space and detect malfunctions in robotic software modules. The analysis of outputs includes the computation of the first and second derivative functions that can unveil anomalies in the behavior of the software. We implemented this methodology in *DiscoFuzzer*, the discontinuity-based fuzzer for ROS, and evaluate three different sampling approaches based on Monte Carlo, Chebyshev, and Spline methods. The discontinuity analysis of *DiscoFuzzer* detected 63 distinct vulnerabilities: 57 of the 89 previously known vulnerabilities, and six new ones. The discontinuity analysis of *DiscoFuzzer* detected 41 more unique vulnerabilities compared to crash detection used by the majority of fuzzers. Furthermore, we found out that no configuration is statistically better in finding vulnerabilities, but each of them discovered vulnerabilities the others could not. However, the `chebfun` configuration occurs to be the fastest in finding vulnerabilities.**

I. Introduction

Robotic systems are proliferating in our society due to their capacity to carry out physical tasks on behalf of human beings. Current applications include, but are not limited to, military, industrial, agricultural, and domestic robots [1]. In this paper, we consider robotic systems as a subset of cyber-physical systems (CPS). While CPS are defined as "physical and engineered systems whose operations are monitored, controlled, coordinated, and integrated by a computing and communicating core" [2], a robot is a "actuated mechanism programmable in two or more axes with a degree of autonomy, moving within its environment, to perform intended tasks" [3]. The standard ISO8373 distinguishes robots further in industrial and service robots according to their tasks.

However, in the deployments of robots, it is vital to consider their safety-critical nature. Indeed, a faulty robot can irreversibly damage the physical environment in which it is operating, including being harmful to human beings. Thus, testing the robustness of robotic systems is crucial to ensure safety.

Fuzzing has become a central tenant of computer security research over the past 20 years. Fuzzing is the repeated execution of the program using inputs sampled from the input space [4]. This repeated execution is performed automatically with the sampling of inputs adhering to different heuristics. Usually, fuzzers guided by code-coverage [5]–[9] or by symbolic engines dominate the scene of the security community [10]–[13]. Furthermore, most current fuzzers solely focus on traditional software crashes or well-known vulnerable code behaviors with the help of sanitizers [14]–[16].

However, robotic systems have to face different threats that current fuzzers do not adequately address. In this paper, we consider external security threats that consist of an attacker gaining control of specific modules of a robotic system or interfering with the robot's sensors input data [17]–[20]. An attacker can lead a robotic system to disregard safety measures, behave in an unsafe way, or crash.

We propose **a novel fuzzing methodology that exploits the continuity of the physical world to automatically explore the input space and detect malfunctions in robotic software modules**. Our solution improves over conventional software testing because it directly addresses failure states, *i.e.,*unsafe behavior, that other fuzzers do not have the ability to address, and provides a mathematical approach to provide dynamic solutions. The fuzzer operates by performing a single test against a target and constructing a model of the targets' response behavior, including discontinuity analysis. We show how a comparison of these tests on several inputs can detect anomalies that are usually only unveiled by extensive manual testing.

Our main contributions in this paper are:

- A novel fuzz testing methodology to examine robotic software systems based on numerical analysis and discontinuity localization;
- A formalized benchmark for ROS consisting of 14 popular open-source packages and 89 previously known vulnerabilities (plus eight new vulnerabilities discovered by *DiscoFuzzer*);
- The design, implementation, and evaluation of *DiscoFuzzer*, its discontinuity-based detection mechanism, and three different sampling approaches.
- The detection of 8 new vulnerabilities in 7 packages included in the benchmark never reported before; 5 have been officially confirmed as potential security threats and added to the Robot Vulnerability Database (RVD) at the time of writing.

We demonstrated that discontinuity analysis could detect more than three times the vulnerabilities that the crash detection mechanism finds. Both *DiscoFuzzer* and the used benchmarks are publicly available on GitHub for the reproducibility of the experiments and to aid further research in automated testing of ROS packages[1].

[1]The URL is not available due to the double-blind process

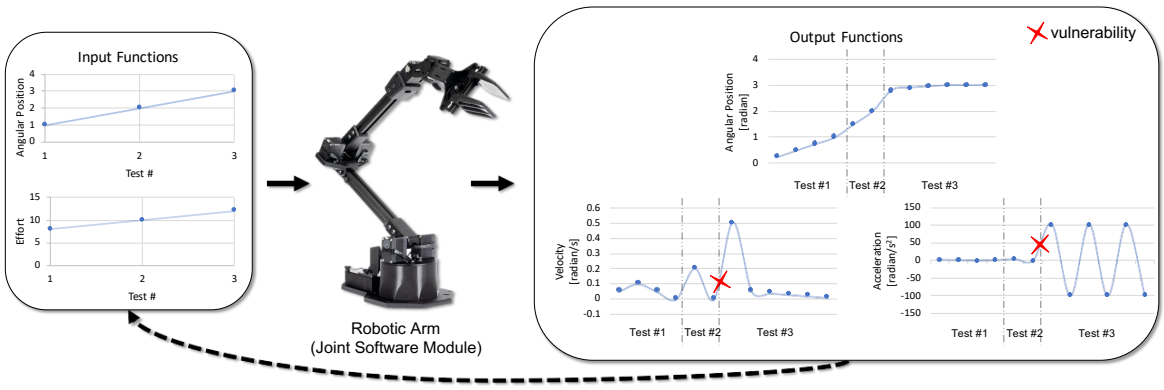


Fig. 1: Example of discontinuity-based vulnerability detection for the software module of a robotic arm's joint.

II. Discontinuity-Based Fuzz Testing

Let us consider an industrial robotic arm with a single joint. The joint software module receives as input the desired angular position and effort (power) to provide to the joint motor to perform the movement. The output is an array describing the actual change of angular position, velocity, and acceleration in time as performed by the joint. Fig. 1 shows an example of how the proposed methodology works. The discontinuity fuzzer begins by first choosing a range of continuous input values. In this case, the range consists of three values for the angular position and three values for the effort. They constitute three fuzz test cases that are provided one by one to the joint software module of the robotic arm. Each defined test case wants to spin the joint one radian clockwise but with increasing effort, *i.e.,* the power to provide to the motor to perform the movement. After each test, we collect the output messages and concatenate their values to create the output functions. The fuzzer then computes first and second derivative functions to analyze discontinuities and unveil anomalies in the behavior of the software. Indeed, the output functions of velocity and acceleration contain a significant change of the slope. The fuzzer detects an anomaly that consists of the uncontrolled rapid rotation of the joint, which could lead to a burnout of the motor because the software did not check the physical constraint The methodology reports the properties of the output function to help guide input test-case generation.

This example illustrates the functionality of discontinuity fuzzing. Initially, our methodology identifies all of the target's inputs and outputs. For each input, it creates a sample generator to provide intervals, *i.e.,* ranges of continuous values, depending on the input types (including arrays). Regarding floating points, the sample generator also needs to define the distance between values in the interval (*i.e.,* the resolution). The methodology changes one input at a time while setting the others to default values. Thus, a fuzzer should perform the following actions:

- Choose an interval: The fuzzer samples a range of continuous values and generates this iteration's test cases (§II-A).
- Test values in the interval: The fuzzer performs one test at a time against the target and saves the outputs.
- Analyze the output distribution: The fuzzer analyzes all of the outputs of this iteration to detect anomalies (§II-B).

The fuzzer iterates on these actions until a termination criterion is met, such as a time limit.

In the following subsection, we present some approaches we adapted to implement the above methodology.

A. Sampling approaches

A sampling generator should be chosen for the first step of the proposed methodology. A sampling generator consists of an iterative process to choose the next group of samples to test. We adapted three sampling approaches based on the authors' expertise and discussion with robotic developers in the industries.

Monte Carlo Sample Generator. Monte Carlo is a method based on repeated random sampling. The sample generator chooses a random value to be the central point for evaluation, and N values are sampled in each direction for derivation analysis, for a total of 2N+1 values per iteration. For arrays, the fuzzer randomly chooses between (1) creating the array repeating a single sample point, and (2) sampling a central point and creating the array with increments of the central point. Within this paper, we shall refer to Monte Carlo sampling as `monte_carlo`.

Chebyshev Sample Generator. Chebyshev is a numerical method that generates increasingly complex polynomials (Chebyshev polynomials) for the interpolation of a dataset. One of the vital functions that Chebyshev polynomials provide is the efficient location of roots and inflection points, computed at each approximation. The Chebyshev sample generator initially returns a list of interpolation inputs. Once the target runs the interpolation inputs, the new list of interpolation inputs is generated based on the target outputs. Indeed, this approach recursively determines which points would be most optimal to sample, given the current function approximation. Within this paper, we shall refer to Chebyshev sampling as `chebfun`.

Spline Sample Generator. Similar to Chebyshev, Spline is

a numerical method that interpolates the function through a piecewise polynomial called a spline. Spline sampling begins with the generation of two clusters of random values, in the same way as `monte_carlo` does. Then, it generates a cluster at the midpoint between these two. If the output from the third selected cluster is similar to the spline interpolated output of the first two clusters, within a user-specified tolerance, then a spline is drawn between the first two clusters. Thus, it randomly generates a new cluster. Otherwise, it generates a new cluster within the bounds set by the first two generated clusters, and it performs a new interpolation. For arrays, it follows the same behavior as `monte_carlo`, with the caveat that the methods for filling arrays are treated as two distinct sources of input for interpolation and generation separately. Within this paper, we shall refer to Spline sampling as `spline`.

Structurally, the `monte_carlo` simulation has the lowest processing overhead and the fastest turnaround, allowing it to cover a larger number of potential inputs. However, the `chebfun` analysis provides a more precise analysis. The `spline` approach acts as a hybrid of the two, with moderate precision and processing overhead.

B. Discontinuity Analysis approach

Different analyses can be done on the output distribution for the last step of the proposed methodology. We believe discontinuities in the output functions are the most significant source of issues as the difference in state course changes incredibly quickly as the system runs. The discontinuity analysis approach first looks for discontinuities in the output. It focuses on sharp changes in the slope of the output functions: large values in the first derivative indicate a significant change, and large values in the second derivative indicate a significant effect. We believe that these changes in the derivatives are more likely to be indicative of errors than any other behaviors, as it is less likely for a numerical output with a smaller average derivative change to be the source of vulnerability.

After some explorative experiments, we realized that as cyber-physical systems tend to maintain a consistent internal state about the world which they use to make decisions, our approach should also consider the amount of influence past inputs have on current outputs. This issue is more obvious to recognize with the input of large floating-point values: these values continue to affect the output regardless of the new provided value due to the mathematical rounding properties. Once an input corrupts the internal state, unexpected or malicious behavior becomes more likely in the long term. Thus, the analysis raises an error when the relationship between input and output becomes significantly different from the history of the system (see section IV)

Furthermore, we choose to look for anomalies where changes to one input cause changes in multiple outputs. We enhance the analysis by considering how many times a field in the output topic has changed. If it changes at some consistent rate, then there is presumed to be a connection between the published topic and the output field. However, if there is a one-off outlier, it is more likely to be a memory or mathematical calculation error. For example, consider a memory-overflow vulnerability. If an input makes the software write beyond the bounds of an array, it could overwrite the other inputs. Thus, a field could change in the output response message when it would typically not be affected by that type of input.

In summary, our discontinuity analysis focuses on two main patterns: values changing very quickly and values changing unexpectedly.

III. Implementing for ROS: Challenges

The Robotic Operating System (ROS) is a collection of software libraries that enables communication of both (abstracted) hardware and (pure) software components to develop robotic systems. It is predicted that ROS-centered systems will make up the majority of robotic systems within the next five years, both in commercial and academic settings [21]. Even among non-ROS systems, many design similarities exist wherein similar methods can be applied [22]. ROS is an open-source project, and it has a vibrant community with thousands of developers and over nine thousand unique packages available[2]. Usually, these packages extend ROS core functionalities by implementing commonly used robotic software modules such as hardware drivers, robot models, datatypes, planning, perception, localization, simulation tools, and other algorithms. Of particular mention is the ROS-Industrial [23] consortium: it consists of 78 international industries that chose to extend and adopt ROS for their industrial robots.

In ROS, independent computing processes called nodes communicate through messages. Messages define clean and consistent interfaces. A **ROS node** is a self-contained process that controls a part of the robot's operation. A **ROS topic** is an implementation of a channel in a publish-subscribe model. Topics act as a named bus where nodes can join as either a publisher, a subscriber, or both. When a publisher node sends a message over a topic, every subscriber node to that topic receives a copy.

The ROS project already includes guidelines for testing [24] by applying existing libraries, *i.e., gtest* for C nodes and *pytest* for Python nodes, for simple unit testing. A tester should design and implement every test in the proper language with test inputs and oracles for every behavior they want to check. Santo *et al.* [25] extended ROS unit tests with property-based testing. Their tool randomly and automatically generates test inputs while checking two simple properties of the node: liveness (a node should not crash) and interface stability (the set of topics should not change). However, adding new properties requires formal reasoning about the target node to formulate its specific correct behavior.

Thus, we recognize the following challenge: designing a ROS fuzzer to minimize human effort, explore the input space automatically and efficiently, and detect any deviation from correct behavior.

[2]https://metrics.ros.org/

IV. DISCOFUZZER

DiscoFuzzer implements the presented novel methodology to target ROS nodes. The fuzzer is developed in Python 3.6 and targets ROS kinetic.

Fig. 2 shows an overview of *DiscoFuzzer*. The fuzzer's input is a user-defined configuration file that includes target information. It outlines the type of messages used in the topics published or subscribed to by the target node. It defines the input topic for the sampling generators. The configuration file also contains how to execute the target node, *i.e.,* flags, and arguments for the command-line. An orchestrator reads the configuration file and it ① initializes the ROS environment with the target node and *DiscoFuzzer*'s publisher and subscriber. A timer starts to count how long the fuzzer should test the target. Then, the first iteration starts. The orchestrator ② notifies the test case generator of the new iteration. The generator samples a new interval, and it ③ provides one value at a time to the ROS environment through the publisher. Once the subscriber receives a message, it ④ copies this value into the output analyzer. The analyzer ⑤ sends the result of the analyses back to the orchestrator. When anomalies are detected, the orchestrator ⑥ relaunches the node and returns the node to a starting state so that testing can continue. The fuzzer's output is a set of reports issued during the fuzzing campaign.

A. Test case generator and Publisher

The test case generator takes the current state of the system and generates a new set of inputs to test through the publisher. The user can define the *interval size*, *i.e.,* the number of inputs to generate at every iteration. If the sampling generator computes the size algorithmically (*e.g.,* in `chebfun`), it ignores this configuration parameter.

The generator tracks which inputs it has already given to the system to ensure that there are no unnecessarily repeated values. Beyond that, it maintains a collection of every single combination of potential inputs from the topic to publish on. This collection allows us to fuzz individual inputs one at a time, as well as inputs in groups of size $s_2 ... s_n$, where n is the size of the input. It allows for the identification of potential interactions or discontinuity in situations where the combined two inputs can break the system, even if they could

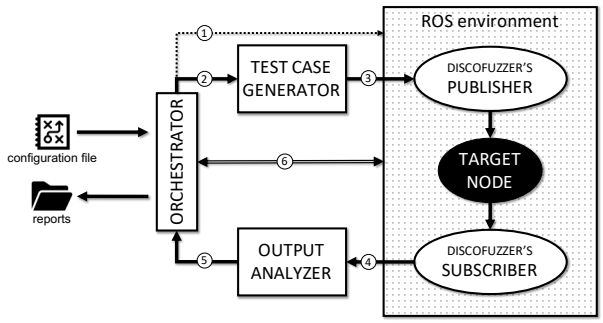


Fig. 2: Overview of DiscoFuzzer

not individually. It focuses on single inputs and implements a power-law drop-off for values fuzzed in combination, prioritizing potentially interesting interactions found from the single fields before conducting a random search. For every ten values fuzzed, nine values are single, and one is a combination. For the combinations, 90% are pairing, 10% are combinations greater than two.

The test generators implement the three sampling approaches presented in the methodology.

For the **`monte_carlo`** sampling approach, the fuzzer chooses the central input at random. The fuzzer stores every point, and it discards any new point that overlaps with the previous range. Additionally, we chose to initialize the fuzzer with some common failure-inducing values such as `NaN`, infinity, and very large or very small floating points.

Since the **`chebfun`** approach works by calculating a polynomial and measuring the result, the generator relies on the information provided by the output analyzer (through the orchestrator). For our implementation, we chose to use the `pychebfun` library [26] as it had the most active development and stars on GitHub outside of the Matlab core implementation. As the `chebfun` approach stores only the expansion coefficients we do not require any space-saving optimizations for a longer run.

The **`spline`** approach consists of a numerical analysis that stores a group of splines containing both measured points and their derivatives, allowing for more accurate interpolation. One of the main benefits of this method is that it allows arbitrary N-dimensional interpolation of functions without extra computation. The `spline` interpolation allows us to perform a more accurate analysis of array fields. For our implementation, we rely on the `scipy` interpolation library [27], choosing our points for the initial array in a random method, just like the `monte_carlo` approach. The `spline` interpolation approach is refined with a mixture of midpoints and random additional points.

For generating an interval of floating points, *DiscoFuzzer* requires the user to choose a *resolution*. The resolution both represents the distances between points in the same interval and indicates the number of decimal digits to consider.

The rate of publishing is limited in two areas: (1) the rate at which the targeted node can respond and (2) the rate at which the subscriber can update. While we can optimize the subscriber and assign new threads to it, there is nothing we can do about the targeted node. As such, the publisher runs on a single thread, and no additional processing capabilities are required.

Users can also choose to set limitations on the `monte_carlo` sampling if they expect that only specific values will produce interesting results. Users can set a *compression factor* and/or place *constraints on combinations*.

In the first case, *DiscoFuzzer* will choose a range at random with size equal to the compression factor with bounds determined by the *resolution*. Then, *DiscoFuzzer* sweeps through the range from bottom to top at the chosen resolution, only saving the center of the range instead of saving each selected

random point. Effectively, whenever *DiscoFuzzer* chooses a value for its sampling, it chooses the next value in the specified range instead of a completely random value. Once the range has been fully explored, it will randomly select another range and repeat the sweeping process. This gives users far more space-efficient storage for their simulation at the cost of far fewer random values being explored over time. As an example, suppose the user sets a range size of 100. When using the *compression factor* feature, after sampling 10,000 points it would only have 100 values saved in memory instead of the 10,000 total values it would have under normal sampling procedures.

The *constraints on combinations* feature allows users to set constraints on generated inputs. These can vary in complexity from simple constraints such as limiting fields to a specific range of values (such as $x < N$), to specific combination rules (such as $y > x; y < N$). Users can specify any numeric constraints they deem necessary for their purposes. This allows users to better guide fuzzing processes to be more efficient.

B. Output analyzer and Subscriber

The output analyzer takes the outputs captured by the subscriber and analyzes them to provide reports to the orchestrator.

The analyzer operates by creating a mapping of potential discontinuities and their inputs. It begins by establishing a baseline for every single output topic, by looking for integer, floating-point, Boolean, and array outputs, and stores a shortened differential vector for them. First, it collects the five (for `monte_carlo` and `spline` sampling approaches) or N (for `chebfun` sampling approach) values. Then, it starts the anomaly detection process in a separate thread while signaling to the orchestrator to send a new group of values. This ensures that the system is utilizing as much of the duty cycle as possible.

The anomaly detection algorithm is at the center of this concept. We primarily rely on numerical approximation rules, as well as analysis of previous research on bugs behavior in cyber-physical systems. Our criterium consists of (1) numerical discontinuities *i.e.,* points where the difference between two values is significant and nonlinear, and (2) information leakage across values in a message, *i.e.,* when a field that is previously unchanged suddenly changes after input changes, especially when it is an outlier or a unique instance.

The discontinuity measures come from the underlying assumption of the continuous nature of the physical world. While there are valid reasons for discontinuity in the output of a robot, if we notice large values in the first or second derivative, mainly when such values are not reflected in their neighboring value derivatives, it necessitates further investigation. These discontinuities are often a sign of a bug in the control loop or floating-point mathematical computations. The quintessential example of this is a division that is rapidly approaching infinity as its denominator tends toward zero. We use the first derivative to look for any logical jumps, and the second derivative to look for any potential significant changes in the first derivative. The analyzer records the mean and the standard deviation for the first and second derivatives of each output. Any deviations more than a *detection threshold* are raised as a potential anomaly.

Furthermore, the analyzer includes a Bayesian sensitivity analysis [28] that calculates the weighted effect of each of the past *N* inputs on the output. The weighted effect is calculated by measuring the derivative effects of the output from a given group of inputs. This weighted effect is stored in an array of size N, and the sum of this array is always one. The weight array is continually updating, maintaining size N. Empirically, we chose to consider the first 50% of values as "old", and the second 50% of values as "new". The analyzer raises an anomaly if the sum of the "new" values is less than the *weighting threshold*. This indicates the process is no longer updating that output for new inputs, flagging a potential anomaly.

We consider `NaN` and infinity outputs as leading special cases of discontinuity. These values are vital for analysis, given that once an internal robotic state system starts to propagate `NaN` values, it is highly likely that it will continuously output `NaN` values. Unless the system has proper handling functionality, the same applies to values of infinity. We only flag `NaN` and infinity values when they occur in more than three consecutive listings, as this was determined to be a sign that the robot is unable to perform any new calculations.

The analyzers also look for anomalies where changes occur unexpectedly in some outputs. Upon startup, a mapping is created for every input and output field combination. This mapping is one-to-one or one-to-many. When a new message is received from the subscriber, for every field changed a counter is incremented for those particular mappings. The analyzer raises an anomaly when the counter is incremented, and the counter value is below a *normality threshold*, a user-defined ratio of the maximum difference between any two mappings for the same input field. A low counter indicates that a value has changed when it normally remains unchanged, a potential anomaly.

By default, all three thresholds (*detection*, *weighting*, and *normality*) are set to be equal to two σ or two standard deviations from the mean. This value is based on the idea of using a 95% confidence interval for fuzzing anomaly detection from Zhao *et al.* [13]. Each threshold can be customized by the user. For example, a user can specify the *detection threshold* and *weighting threshold* at two σ, but set the *normality threshold* to three σ, if an output field has a low rate of change causing a larger number of false positives than expected.

Once the anomaly detection algorithm has discovered an anomaly, the analyzer saves inputs, outputs, and timings of the test, as a pickled message object.

C. Crash Detection

DiscoFuzzer implements another anomaly detection mechanism, performed by the orchestrator, that fuzzers use typically: crash detection. The orchestrator deploys the ROS nodes in python subprocesses. When a node crashes, the subprocess

returns with an error code that is caught by the orchestrator. Furthermore, the orchestrator periodically uses the utility *ROStopic* [29] to check whether the topics are still available to the nodes. When a crash is detected, it creates similar reports every time it discovers an anomaly.

D. Source Code

The source code of *DiscoFuzzer* is publicly available at URL[3].

V. Evaluation

In this section, we evaluate *DiscoFuzzer* for the effectiveness of the discontinuity-based analysis approach and the efficiency of the three sampling approaches.

More specifically, we conduct an experimental campaign to answer two main research questions:

RQ1 Can discontinuity analysis detect novel vulnerabilities in ROS nodes? (§V-C)

RQ1.1 How many bugs does the discontinuity analysis detect compared to the other analyses, previous work and human detection?

RQ2 Which sampling approach is more efficient to trigger vulnerabilities in ROS nodes? (§V-D)

We initially create a set of benchmarks by including popular ROS nodes with known vulnerabilities (§V-A). Then, we execute *DiscoFuzzer* against the benchmarks (§V-B). Furthermore, we also consider some common use-cases and threats to validity to conclude the evaluation(§V-E,§V-F).

A. Benchmarks

We identified 14 total open-source ROS packages in the ROS ecosystem, using the rosmap [30] tool. Ten of them are the packages with the highest number of Github stars, contributions, and packages that depend on them, *i.e.,*those packages with very high impact on the community. These ten core packages are usually included in academic robotic systems to support research, development, and prototyping [31]. The additional four are real-world packages used in non-academic contexts. These packages are highly rated on GitHub: the Autoware self-driving car, the Udacy self-driving car, the NASA Mars rover, and the ARDrone flight system.

All code for the packages is hosted and regularly updated on GitHub. For every package, we looked into its issue tracker for closed issues. We kept only those issues related to software vulnerabilities and discarded the others (i.e. users' questions, feature requests, or compilation issues). We assign to each vulnerability a numeric and unique id. We define a benchmark as the code in the package's repository at a version that includes the maximum number of vulnerabilities not already included in other benchmarks. This criterion resulted in more benchmarks for the same package but at different versions. Thus, we identified 20 benchmarks with 89 vulnerabilities.

The full information on the benchmarks can be found at https://anonymous.4open.science/r/ISSREAppendix-7C74/benchmarks.pdf.

[3]The source code will be made available upon acceptance on Github.

During the execution of fuzz campaign, we detected further vulnerabilities in the targets. For each of them, we extensively searched into the issue trackers and look for the related issue. If not found, we consider the new vulnerability as previously unknown to the community and we submitted a new issue related to it. We assigned to all of them a numeric and unique id, and we added them to the *DiscoFuzzer*'s benchmarks. Since we reported the 8 vulnerabilities to the developers, and have been officially confirmed as potential security threats. Two of these remain open, while the others have been addressed.

B. Experimental design

DiscoFuzzer's configuration parameters were experimentally tuned with for all three different sampling approaches. Table I shows the values of all other parameters that are identical among the three. These parameters were determined experimentally after exploratory analysis with *DiscoFuzzer*.

For each combination of a benchmark and a sampling approaches, we run the fuzzer 10 times. A single repetition, *i.e.,* a fuzz campaign [4], lasts 24 hours. The only exception is the `chebfun` sampling approach: it only needs to run once per benchmark because of its deterministic nature. In total, we ran 420 fuzz campaigns, 420 full days in CPU time.

The fuzzing environment runs in a virtual machine with four cores of an Intel(R) Xeon(R) CPU E5-4650 set in a host emulation, 8 GB of RAM, Ubuntu 16.04, ROS kinetic, Python 3.6. We optimize the execution of the program with the python profile library and implement an automatic load balancer between the subscriber and the publisher to ensure that the system is constantly publishing at the highest rate that the fuzzed node can support. We enforce the testing time by using the system clock and terminate the program after 24 hours.

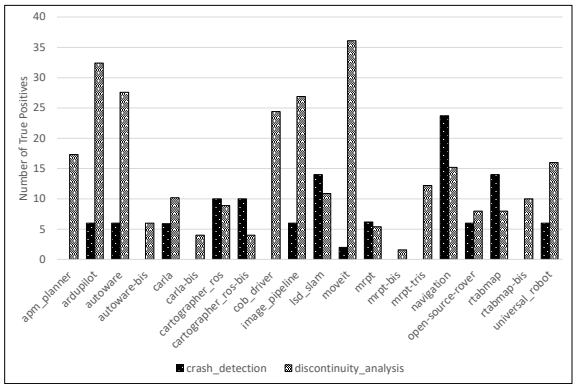
C. Effectiveness of DiscoFuzzer

Every fuzz campaign produced a report for each vulnerability detected by *DiscoFuzzer*. The report contains information such as sampling approach, target node, last messages (i/o), anomaly type, and detection time (elapsed time since the beginning of the fuzzing campaign). We manually inspected every report and tested its reproducibility. We labeled the report with its vulnerability-id if it produced a True Positive.

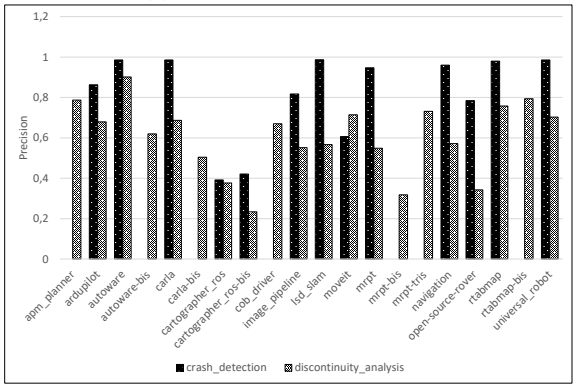
DiscoFuzzer found eight previously unknown vulnerabilities, and identified 77 of the 89 previously known vulnerabil-

TABLE I: *DiscoFuzzer*'s configuration parameters used in the evaluation.

parameter	value
interval size	5
resolution	0.01
compression factor	200
detection threshold	2 times the standard deviation of the expected values
weighting threshold	3 times the standard deviation of the expected values
normality threshold	2 times the standard deviation of the expected values

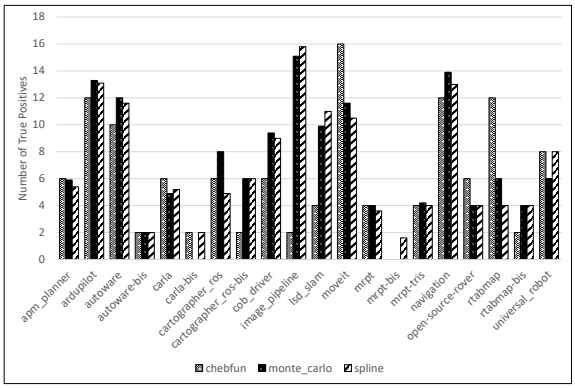


(1) Number of True Positives

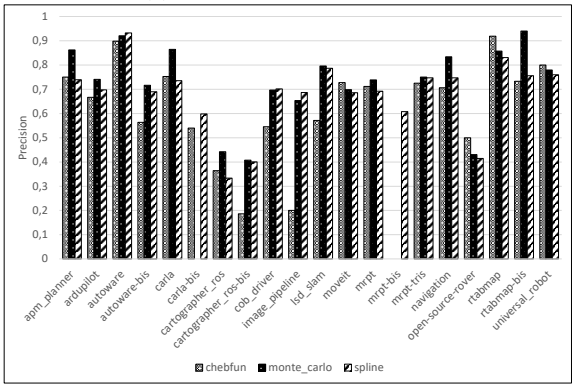


(2) *DiscoFuzzer*'s Precision

Fig. 3: Metrics grouped by the issuing detector mechanism. The bars' height represents the mean value for the group in the specified benchmark.



(1) Number of True Positives



(2) *DiscoFuzzer*'s Precision

Fig. 4: Metrics grouped by the used sampling approach. The bars' height represents the mean value for the group in the specified benchmark.

ities.

We grouped the reports by the issuing detector, *i.e.,* either `discontinuity_analysis` or `crash_detection`. The crash detection mechanisms detected 22 distinct vulnerabilities: 20 of the 89 previously known vulnerabilities, and 2 new ones. During the evaluation, the discontinuity analysis of *DiscoFuzzer* detected 63 distinct vulnerabilities: 57 of the 89 previously known vulnerabilities, and 6 new ones (*cfr.* **RQ1**).

Fig. 3 shows the number of true positives and the precision. Even if the `discontinuity_analysis` can detect more vulnerabilities in 12 out of 14 benchmarks, its complexity and infancy are highlighted by looking at the precision. The `crash_detection` almost always has higher precision than `discontinuity_analysis`.

The discontinuity analysis of *DiscoFuzzer* detected 41 more unique vulnerabilities compared to the crash detection (*cfr.* **RQ1.1**).

D. Efficiency of the sampling approaches

Table III and Fig. 5 present the vulnerabilities as detected by the three sampling approaches. Every sampling approach found unique vulnerabilities that the other two could not. In particular, the `chebfun` sampling approach found more than double the amount of these unique vulnerabilities, including two previously unknown vulnerabilities. In total, eight new vulnerabilities were detected, five of which were detected by all three methods, two which were identified by `spline` and `monte_carlo`, and two that were identified only by `chebfun`.

We grouped the reports by the used sampling approaches, shown in Fig. 4. Following the methodology presented by Klees *et al.* [32], we performed a statistical analysis to determine which of the sampling approaches has better precision. We performed the Mann Whitney U test [33] to test the null hypothesis H_0 that the precision of two sampling approaches is statistically equal. In case of $p\_value < 0.05$, we reject H_0. We computed the Vargha and Delaney's A statistic [34] to compute the effect size of the statistically different group. Given two groups, $A_{1,2}$ is close to 1 if the first group has statistically higher precision than the second group, close to 0 on the opposite case. Table II shows the results of this statistical analysis. The precision of the three sampling methods are statistically different in 14 cases where none always prevailed on the other. In the other cases, the statistical tests are inconclusive. Thus, we did not find a sampling method that statistically improve the overall precision of *DiscoFuzzer*.

Similarly to the precision analysis, we analyzed the vulnerability detection times of the three sampling approaches

TABLE II: Results of the statistical analyses on precision performed among *DiscoFuzzer*'s three sampling approaches, namely `spline` (s), `monte_carlo` (mc), and `chebfun`(c). The p is the p-value of the Mann-Whitney U test performed, while the A is the Vargha and Delaney's statistic. The first column lists the benchmarks' names.

Benchmark	$p_{s,mc}$	$A_{s,mc}$	$p_{s,c}$	$A_{s,c}$	$p_{mc,c}$	$A_{mc,c}$
apm-planner	<**0.001**	0.19	<**0.001**	0.5	<**0.001**	0.90
ardupilot	<**0.001**	0.33	<**0.001**	0.5	<**0.001**	0.90
autoware	**0.023**	0.40	**0.023**	0.7	**0.023**	0.77
autoware-bis	0.079	0.59	0.079	0.75	0.079	0.68
carla	**0.048**	0.22	**0.05**	0.40	**0.05**	0.72
carla-bis	<**0.001**	1	<**0.001**	0.56	<**0.001**	0
cartographer-ros	**0.002**	0.07	**0.002**	0.44	**0.002**	0.89
cartographer-bis	<**0.001**	0.45	<**0.001**	0.99	<**0.001**	0.99
cob-driver	<**0.001**	0.50	<**0.001**	0.9	<**0.001**	1
image-pipeline	<**0.001**	0.66	<**0.001**	1	<**0.001**	1
lsd-slam	<**0.001**	0.53	<**0.001**	1	<**0.001**	1
moveit	0.2214	0.39	0.221	0.2	0.221	0.4
mrpt	0.278	0.5	0.278	0.45	0.278	0.42
mrpt-bis	**0**	1	**0**	1	**0**	0.5
mrpt-tris	0.149	0.40	0.149	0.52	0.149	0.64
navigation	<**0.001**	0.20	<**0.001**	0.6	<**0.001**	1
nasa osr	<**0.001**	0.40	<**0.001**	0.2	<**0.001**	0.1
rtabmap	0.375	0.19	0.375	0.19	0.375	0.46
rtabmap-bis	**0.037**	0.36	**0.037**	0.59	**0.037**	0.73
universal-robot	0.219	0.49	0.219	0.5	0.219	0.4

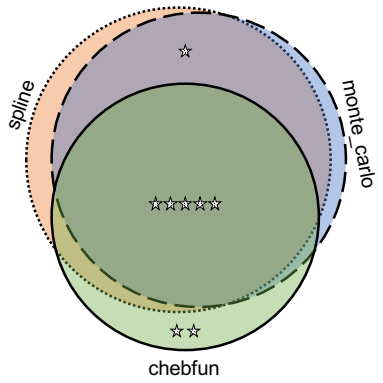


Fig. 5: Venn diagram of the vulnerabilities as detected by the three sampling approaches. The area occupied by the three circles represents the number of unique vulnerabilities found. Overlapped areas indicate the number of unique vulnerabilities found by both (or the three) approaches. The stars represent unique vulnerabilities that were previously unknown.

through statistical tests. The null hypothesis H_0, for the Mann-Whitney U test, is that the detection times for a vulnerability in two sampling methods are statistically equal. The Vargha and Delaney's $A_{1,2}$ statistic is high if the first group has a longer detection time than the other, low in the opposite case. We then compared each sampling approach with the other two. For each comparison, we computed how many times it is statistically faster than the other. Results of the statistical analysis on vulnerability detection times are reported in this Github repository: https://anonymous.4open.science/r/ISSREAppendix-7C74/statistical_analysis.pdf. Table IV lists

TABLE III: Vulnerabilities as detected by the three sampling approaches. The first three columns indicate whether the row is the intersection (●) or not (○) of the elements detected by `spline`, `monte_carlo`, and `chebfun`. The fourth column has the number of vulnerabilities in the resulted subset. The last column contains the ids of the vulnerabilities, highlighting in bold the previously unknown vulnerabilities.

`spline`	`monte_carlo`	`chebfun`	#	vulnerability IDs
●	●	●	46	1, 2, 7, 8, 9, 10, 12, 13, 15, 16, 17, 21, 23, 26, 27, 34, 35, 39, 40, 53, 54, 55, 56, 57, 59, 60, 61, 62, 64, 66, 67, 68, 72, 73, 74, 81, 82, 86, 88, 89, **90**, **92**, **93**, **96**, **97**
●	●	○	21	11, 14, 20, 25, 29, 32, 33, 36, 37, 41, 42, 43, 44, 45, 47, 49, 50, 52, 70, 83, **91**
●	○	●	4	18, 24, 71, 85
○	●	●	3	19, 28, 80
●	○	○	3	48, 52, 63
○	●	○	2	46, 65
○	○	●	7	22, 30, 75, 77, 79, **94**, **95**

TABLE IV: Pairwise comparison among the vulnerability detection times of three sampling approaches of *DiscoFuzzer*. While the second column present us with the results based on statistical significance, the last column presents the number of ties. There is a tie in a comparison of vulnerability detection times if such a comparison is not statistically significant, *i.e.,*the p-value of the associated Mann-Whitney U test is greater than 0.05.

Comparison	Result	Ties
`spline` - **`monte_carlo`**	15 - 25	26
`spline` - **`chebfun`**	11 - 19	19
`monte_carlo` - **`chebfun`**	13 - 17	18

the results of these comparisons, demonstrating that `chebfun` approach is statistically faster to detect bugs.

No sampling method is statistically better in finding vulnerabilities, but each of them discovered vulnerabilities the others could not. However, the `chebfun` approach occurs to be the fastest in finding vulnerabilities (*cfr.* **RQ2**).

E. Discussion

DiscoFuzzer demonstrates its capabilities in detecting previously known and unknown erroneous behavior. The three sampling approaches complement each other well to identify many types of cyber-physical vulnerabilities. In the majority of cases, *DiscoFuzzer* successfully identified vulnerabilities with high precision. However, testing revealed some case-specific behaviors that are notable for future analysis.

When considering the security of ROS systems, there are three viewpoints to analyze. First of all, ROS systems are insecure and provide little protection against attackers [17]. The second point of view is that through leveraging all security modules available through ROS [35], we can treat these

systems as reasonably secure: the only concern is sensor-based attacks. As an extension of the second viewpoint, the third one only considers attacks that could potentially leave long-lasting or hard to repair damage to the physical systems or surroundings. *DiscoFuzzer* operates under the assumption that the only vulnerabilities in ROS systems concern sensor-based attacks. Thus, every anomaly found by *DiscoFuzzer* is not adequately protected by the current security measures and can be exploited to cause damage to the robotic system and its surroundings. In this way, *DiscoFuzzer* provides an important layer of protection.

Overall, the false positives are exceedingly rare for crash failures. Their primary cause is the use of asserts in the ROS code. Notably, Google Cartographer exhibits a different crash profile as Google Cartographer uses asserts while parsing messages. For any invalid message, the assert causes the node to cease execution. After a manual analysis of the design and code of the node, we hypothesize that this is a design decision made to enforce well-formatted inputs. Even applying constraints and automatically eliminating any combination that caused too many crashes, the tool still picks up a moderate amount of false positives. However, it is necessary to note that we chose a consistent threshold for testing all packages for this paper. In real-world applications, users can tune the fuzzer for their specific applications. Additionally, users can easily construct filters to cut down on the false positives, as each node has a unique false-positive profile.

Here, we found that false negatives are primarily due to specific configuration requirements or bugs that required multiple fields to be at precise values.

It is important to note that the packages chosen to analyze *DiscoFuzzer*were the most used in the ROS ecosystem, measured by stars, dependencies, and contributors on Github. As these packages generally have the most developer attention, it was expected that it would be difficult to identify novel vulnerabilities or software issues that were not already accounted for through active development. To demonstrate the power of *DiscoFuzzer*, we also tested the lesser-used packages[4]. We discovered an additional 15 novel bugs among only three packages. From these results, it is clear that *DiscoFuzzer*can identify vulnerabilities in a variety of packages.

A limitation of *DiscoFuzzer* is data-type selection. *DiscoFuzzer* only analyzes numeric data types, such as integers, floating points, boolean values, and arrays. Other data types are, therefore, excluded. While our approach behaves well for our stated goals, it deliberately excludes vital data types such as strings, a typical application in fuzzing research. A future version of the tool can mitigate this limitation by using a combination approach with other well-established non-numerical fuzzers such as AFL [36].

While previous work on property-based fuzzing is valid, *DiscoFuzzer* outperforms them in detecting vulnerabilities that combine multiple properties in unison, which traditional property-based fuzzing detects less efficiently [25]. Additionally, *DiscoFuzzer* is more accessible for developers to validate discovered vulnerabilities. Similarly, another recent advancement in the field includes derivative-based fuzzing [37]. However, this approach is narrow in scope, focusing only on the control loop of systems. *DiscoFuzzer* is capable of a broader scope in its analysis, able to identify a more significant number of potential vulnerabilities with applications among any ROS package.

[4]the 80th percentile of packages in the ROS ecosystem measured by the metrics above

F. Threat to validity

The only threat to validity that we are aware of is external. As we focused our initial design on ROS, we did not guarantee that our discontinuity results are generalizable to other cyber-physical systems. However, we believe that the approach is general enough to apply to any system with well-defined inputs and output interfaces.

VI. Related Work

Previous research related to our work includes fuzz testing and state-of-the-art robot system testing.

A. Fuzz testing

In recent years, the security community advanced enormously in the state-of-the-art of fuzzers, proposing different approaches to generate new inputs and detect the program's misbehavior. The most recent advancement in derivative-based fuzzing is an approach by Taegyu *et al.* [37], which focused on fuzzing the PID control loop of several types of robots. This approach utilized control loop stability as a means to guide coverage testing for systems that use the MAVlink protocol. kAFL [5] is a coverage-guided kernel fuzzer that is OS-independent and based on hardware-assisted instrumentation. DIFUZE [6] is an interface-aware fuzzing tool to automatically generate valid inputs and trigger the execution of the kernel drivers. JANUS [7] is a feedback-driven fuzzer that explores the two-dimensional input space of a file system, that mutates metadata on a large image while emitting image-directed file operations. PeriScope [8] focuses on the hardware-OS boundary, targeting device drivers. Razzer [9] guides the exploration of inputs to search for data races.

Concolic execution can help the exploration of the input space where standard approaches fail. These fuzzers are called hybrid and vastly showed their efficacy in testing software programs [10], [11], [12], [13]. However, the community also provided lighter methods to explore the program under test and overcome magic numbers and nested checksums. VUzzer [38] leverages control- and data-flow features to generate interesting inputs. TFuzz [39] solves the problem by removing sanity checks in the target program when it cannot bypass them. REDQUEEN [40] exploited the input-to-state correspondence, *i.e.,*the feature of some program where parts of the input often end up directly in the program state. Some fuzzers try to use some level of knowledge of the input space and generate fewer but better inputs. TIFF [41] tags input bytes with its basic type (*e.g.,* 32-bit integer) in the program, then it uses this information to mutate the inputs accordingly. ProFuzzer

[42] adds to this the ability to understand input fields of critical importance. Superion [43] creates meaningful tests for programs that process structured input (*e.g.,*XML engines) by analyzing its grammar.

The differential testing approach is a way to find errors by comparing results from different implementations of the same program, resulting in cross-referencing oracles. Chen *et al.* [44] applied differential testing to a coverage-guided fuzzer for Java Virtual Machines. Nezha [45] exploits these differences also to generate inputs that are more likely to trigger a bug. DifFuzz [46] explores the input space by maximizing the differences in resource consumption since it targets side-channels vulnerabilities.

Sanitizers are another approach to detect specific failures in a fuzzing campaign. AddressSanitizer [14] finds out-of-bounds accesses to the heap, stack, and global objects, as well as use-after-free bugs. UndefinedBehaviorSanitizer [15] detects null pointer, signed integer overflows, and wrong type conversions. MemorySanitizer [16] focuses on uninitialized reads.

The novelty of *DiscoFuzzer* lies in taking the previous model-based testing approaches and extending their most commonly found failure modes into a highly search-based system. We provide a standard extension framework for analyzing model-based bugs. *DiscoFuzzer* is highly user-customizable and aimed at an under-explored area of fuzzer research: cyber-physical systems. As of the publishing of this paper, this is the first-known approach that combines these methods for applications within the cyber-physical domain. Additionally, this approach is the first applied to Robotic systems, namely ROS.

B. Robotic System Testing

Robotic systems are validated with formal specifications, and these formal verification approaches are complementary to the testing of the actual implementation of the robot. With the assumption of functioning hardware, we focus on how to test robotic software.

DiscoFuzzer falls into the category of search-based approaches. A search-based method uses heuristics to explore the input space of the target and automatically generate test cases. Both Ali and Yule [47] and Arrieta *et al.* [48] [49] use genetic algorithms to generate and select test cases. Matinnejad *et al.* [50] applied different search algorithms for testing automotive embedded systems, including random search, adaptive random search, a hill-climbing algorithm, and a simulated annealing algorithm. Abbas *et al.* [51] applies the Monte Carlo simulation to explore the input space to find violations of robustness properties. Instead, *DiscoFuzzer* is not applying Monte Carlo on models nor focuses on temporal logic. *DiscoFuzzer* treats the module as a black-box, and its Monte Carlo implementation is aware of the state of the robot system.

Specific to ROS, RosPenTo is a semi-automated tool for testing ROS [17]. It injects fake messages into the system to demonstrate the consequences of the lack of security in ROS. Indeed, an attacker (1) can easily query the master for sensitive information, and (2) can easily impersonate a subscriber node. These are the assumptions we used for *DiscoFuzzer*. Indeed, *DiscoFuzzer* assumes that the message and packaging interface of ROS are wholly intact and reliable, and focuses on the effects of those messages on the system.

Santos *et al.* [25] implemented a property-based testing framework for ROS. This first approach aims at the automatic generation of test scripts for property-based testing of various configurations of a ROS system. Their approach looks for sequences of messages that either crashes the target node or violate a previously specified property. An automatic test generation method builds the property-based tests from configuration models extracted by a static analysis framework. Instead, *DiscoFuzzer* is a proper fuzzer that, besides automatically generated test cases for the target, does not need any specification of properties to detect anomalies in the target.

VII. Conclusion

In this paper, we have proposed a novel fuzz testing methodology. We chose three different sampling methods and implemented the methodology in *DiscoFuzzer* targeting ROS packages. We tested it against 14 ROS packages to evaluate its efficacy and efficiency. Results show that *DiscoFuzzer* can find more vulnerabilities than traditional crash detection mechanisms without the necessity to specify any ad-hoc property for the targets. However, no sampling approach resulted entirely better than the other, but all of them are necessary to find the majority of unique vulnerabilities. Thus, we envision *DiscoFuzzer* will be able to use a combined approach for sampling and cover all the detected vulnerabilities at once. Furthermore, future research should look forward to improving discontinuity-based analysis and decrease the number of false positives while using the approach.

References

[1] M. Ben-Ari and F. Mondada, "Robots and their applications," in *Elements of Robotics*. Springer, 2018, pp. 1–20.

[2] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: the next computing revolution," in *Design Automation Conference*. IEEE, 2010, pp. 731–736.

[3] "Robots and robotic devices — Vocabulary," International Organization for Standardization, Geneva, CH, Standard, Mar. 2012.

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

[5] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kafl: Hardware-assisted feedback fuzzing for {OS} kernels," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 167–182.

[6] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2123–2138.

[7] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing file systems via two-dimensional input space exploration," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 818–834.

[8] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, "Periscope: An effective probing and fuzzing framework for the hardware-os boundary." in *NDSS*, 2019.

[9] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 754–768.

[10] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, no. 2016, 2016, pp. 1–16.

[11] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.

[12] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 745–761.

[13] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing." in *NDSS*, 2019.

[14] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, 2012, pp. 309–318.

[15] LLVM-Developers. (2020) Undefinedbehaviorsanitizer. [Online]. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[16] ——. (2020) Memorysanitizer. [Online]. Available: http://clang.llvm.org/docs/MemorySanitizer.html

[17] B. Dieber, B. Breiling, S. Taurer, S. Kacianka, S. Rass, and P. Schartner, "Security for the robot operating system," *Robotics and Autonomous Systems*, vol. 98, pp. 192–203, 2017.

[18] S. Rivera, S. Lagraa, A. K. Iannillo, and R. State, "Auto-encoding robot state against sensor spoofing attacks," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2019, pp. 252–257.

[19] S. Lagraa, M. Cailac, S. Rivera, F. Beck, and R. State, "Real-time attack detection on robot cameras: A self-driving car application," in *2019 Third IEEE International Conference on Robotic Computing (IRC)*. IEEE, 2019, pp. 102–109.

[20] D. Davidson, H. Wu, R. Jellinek, V. Singh, and T. Ristenpart, "Controlling uavs with sensor input spoofing attacks," in *10th {USENIX} Workshop on Offensive Technologies ({WOOT} 16)*, 2016.

[21] D. Petrara, "The rise of ros: Nearly 55% of total commercial robots shipped in 2024 will have at least one robot operating system package," 2019. [Online]. Available: https://www.bloomberg.com/press-releases/2019-05-16/the-rise-of-ros-nearly-55-of-total-commercial-robots-shipped-in-2024-will\\-have-at-least-one-robot-operating-system-package

[22] A. Elkady and T. Sobh, "Robotics middleware: A comprehensive literature survey and attribute-based bibliography," *Journal of Robotics*, vol. 2012, 2012.

[23] ROS-Industrial. (2020) Ros-industrial. [Online]. Available: https://rosindustrial.org/

[24] ROS.org. (2020) Automatic testing with ros. [Online]. Available: http://wiki.ros.org/Quality/Tutorials/UnitTesting

[25] A. Santos, A. Cunha, and N. Macedo, "Property-based testing for the robot operating system," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 56–62.

[26] C. Swierczewski and O. Verdier, "Pychebfun," https://github.com/pychebfun/pychebfun, 2020.

[27] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. Jarrod Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. Carey, İ. Polat, Y. Feng, E. W. Moore, J. Vand erPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and S. . . Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, 2020.

[28] R. Weiss, "An approach to bayesian sensitivity analysis," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 58, no. 4, pp. 739–750, 1996.

[29] ROS.org, "rostopic," http://wiki.ros.org/rostopic, 2020.

[30] M. Pichler, B. Dieber, and M. Pinzger, "Can i depend on you? Mapping the dependency and quality landscape or ROS packages," in *Proceedings of the 3rd International Conference on Robotic Computing*. IEEE, Feb. 2019.

[31] A. Martinez and E. Fernández, *Learning ROS for robotics programming*. Packt Publishing Ltd, 2013.

[32] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[33] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[34] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[35] R. White, D. Christensen, I. Henrik, D. Quigley *et al.*, "Sros: Securing ros over the wire, in the graph, and through the kernel," *arXiv preprint arXiv:1611.07060*, 2016.

[36] M. Zalewski, "american fuzzy lop technical "whitepaper","," *URL: http://lcamtuf. coredump. cx/afl/technical_details. txt*, 2015.

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

[38] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing." in *NDSS*, vol. 17, 2017, pp. 1–14.

[39] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 697–710.

[40] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence." in *NDSS*, vol. 19, 2019, pp. 1–15.

[41] V. Jain, S. Rawat, C. Giuffrida, and H. Bos, "Tiff: using input type inference to improve fuzzing," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 505–517.

[42] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, "Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 769–786.

[43] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 724–735.

[44] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of jvm implementations," in *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 85–99.

[45] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 615–632.

[46] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, "Diffuzz: differential fuzzing for side-channel analysis," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 176–187.

[47] S. Ali and T. Yue, "U-test: evolving, modelling and testing realistic uncertain behaviours of cyber-physical systems," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–2.

[48] A. Arrieta, S. Wang, G. Sagardui, and L. Etxeberria, "Search-based test case selection of cyber-physical system product lines for simulation-based validation," in *Proceedings of the 20th International Systems and Software Product Line Conference*, 2016, pp. 297–306.

[49] ——, "Test case prioritization of configurable cyber-physical systems with weight-based search algorithms," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, 2016, pp. 1053–1060.

[50] R. Matinnejad, S. Nejati, L. Briand, T. Bruckmann, and C. Poull, "Search-based automated testing of continuous controllers: Framework, tool support, and case studies," *Information and Software Technology*, vol. 57, pp. 705–722, 2015.

[51] H. Abbas, G. Fainekos, S. Sankaranarayanan, F. Ivančić, and A. Gupta, "Probabilistic temporal logic falsification of cyber-physical systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 2s, pp. 1–30, 2013.