Coverage-Guided Fuzz Testing for Cyber-Physical Systems

Sanaz Sheikhi
ssheikhi@cs.stonybrook.edu
Stony Brook University
Stony Brook, New York, USA

Edward Kim
ehkim@cs.unc.edu
University of North Carolina
Chapel Hill, North Carolina, USA

Parasara Sridhar Duggirala
psd@cs.unc.edu
University of North Carolina
Chapel Hill, North Carolina, USA

Stanley Bak
stanley.bak@stonybrook.edu
Stony Brook University
Stony Brook, New York, USA

ABSTRACT

Fuzz testing is an indispensable test-generation tool in software security. Fuzz testing uses automated directed randomness to explore a variety of execution paths in software, trying to expose defects such as buffer overflows. Since cyber-physical systems (CPS) are often safety-critical, testing models of CPS can also expose faults. However, while existing coverage-guided fuzz testing methods are effective for software, results can be disappointing when applied to CPS, where systems have continuous states and inputs are applied at different points in time.

In this work, we propose three changes to customize coverage-guided fuzz testing methods to better leverage characteristics of CPS. First, we introduce a notion of coverage to be used to evaluate a fuzz testing algorithm's effectiveness for a particular CPS, analogous to often-used code coverage metrics of a software system. Second, this modified coverage metric is used in a customized power schedule, which selects which previous input sequences hold the most promise to find failures in new system states. Third, we modify the input mutation strategy used to reason with the causal nature of a CPS. Our proposed system, which we call CPS-Fuzz, is compared with three other fuzz testing frameworks on a autonomous car racing software and provides a superior coverage score by generating more crashes at different positions around the track.

KEYWORDS

cyber-physical systems, fuzz testing, test generation, coverage metric, autonomous vehicle

1 INTRODUCTION

Cyber-physical systems (CPS) are at the heart of transportation, aerospace, autonomy, healthcare, agriculture, and defense platforms. Since these systems carry out complex tasks, they have a complex structure. Many CPS work in safety-critical environments and are upgraded throughout their lifetimes, where either the software or physical component is updated in subsequent design iterations. Design correctness is an essential element of safety-critical CPS.

Although formal verification approaches for CPS continue to make progress, in practice, due to large scale and high complexity, testing is the primary means to check if a system will be functioning correctly. In software systems, fuzz testing is an indispensable test-generation technique that uses directed randomness to try to explore possible paths through software, in order to detect faults. Fuzz testing approaches have found bugs in compilers [3], flaws in widely-used image parsers [22], security vulnerabilities in web browsers [30], and even soundness violations in formal reasoning tools like SMT solvers [46]. Since the techniques are automated and testing is easily parallelizable, the approaches can be scaled up to run on many machines. Fuzz testing algorithms have been running 24/7 on clusters at large companies like Microsoft and Google for over the last ten years, finding security vulnerabilities and saving the companies millions of dollars[21]. This poses the pertinent question: can fuzz testing also be an effective method for exposing problems in safety-critical CPS?

Unfortunately, generic fuzz testing approaches can be disappointing when applied to a CPS. CPS have unique characteristics, continuous states and inputs that change over time, which make existing coverage metrics insufficient and subsequent search algorithms less effective. In this paper, we seek to remedy this situation by customizing generic fuzz testing approaches to these unique aspects of CPS. The main contributions of this paper are as follows:

- A CPS coverage metric. This metric takes into account the differences in continuous states of two system failures to determine an overall test coverage score, and applies to black-box CPS simulators.
- A customized *power schedule* and *mutation strategy*, two key elements of coverage-based fuzz testing, based on the developed coverage metric, which targets new tests in regions that can increase the coverage score.
- An evaluation of the proposed CPS fuzz testing framework on autonomous racing software, using a simulated environment of the F1TENTH autonomous racing competition.

The rest of this paper is organized as follows. First, we review related work in Section 2. Next, in Section 3, we discuss the basics of the Fuzz testing technique. In Section 4, we present our methodology as a generic test-generation framework that could be used for any CPS. In Section 5, we evaluate our framework and compares it with other fuzz testing methods in an autonomous racing scenario. The paper finishes with a discussion and a conclusion.

2 RELATED WORK

Fuzz testing has become an instrumental technique in the domains of software verification and security analysis. Ever since Miller et al first proposed the methodology to test the reliability of UNIX programs, fuzz testing has become an active field of research finding great efficacy in software verification and security analysis

[33], [36], [48]. Several software companies, such as Google and Microsoft, have developed sophisticated fuzzing tools employing a variety of techniques stemming from white-box, gray-box, and black-box testing approaches [18], [21], [23], [28], [45].

Several works address the fuzz testing of CPS software in order to discover security vulnerabilities [19], [37], [43]. However, there are fewer works investigating the application of fuzz testing in generating test cases for the CPS controller. Recently, Zhong et al developed a fuzz testing technique called *AutoFuzz* to leverage Neural Networks to a guide an evolutionary search over autonomous vehicle API inputs. The network was trained to predict if new seeds will lead to unique traffic violations with the most promising seeds mutated to become new adversarial inputs [49]. However, training Neural Networks could pose a wide range of stability and convergence issues such as the well-known vanishing gradient problem [41].

Hu et al proposed a set of coverage-driven techniques labeled as Autonomous Safety Fuzzing (ASF) [26]. These techniques included new mutation strategies more suitable for testing autonomous vehicles. Some examples include flipping a traffic light color or changing the vehicle's starting position. They also developed a coverage metric, called trajectory coverage, which captures the amount of road area traversed by the autonomous vehicle. To compute the metric, the fuzz tester initially divides the traffic environment into a grid of blocks. During a simulation, the number of blocks crossed by the ego vehicle will be the outputted metric value.

AV-fuzzer [31] is another framework whose main difference is that fuzzing techniques are restricted to particularly risky traffic scenarios where safety violations are highly likely. During an earlier stage, a genetic algorithm combines vehicle kinematic models and safety constraints to find promising scenarios that are likely to lead to a safety violation. The "local" fuzz tester takes feedback from the genetic algorithm, as well as the discovered seed encoding the risky traffic scenario. The fuzz tester explores local perturbations according to a set of mutation strategies under the idea that traffic scenarios with low safety potential could contain a cluster of violations.

Finally, another approach considers Metamorphic Fuzz Testing (MFT) [25]. On a high level, this approach differs from other fuzz testing methods in that the driving environment changes abruptly *during* the simulation phase; the traffic environment is set to delete and impose newly-generated traffic agents every six seconds during the simulation. The authors claim that this designed to examine the vehicle's behavior when the input is invalid or unexpected. To distinguish between genuine system failures, such as planning errors, and false positives, such as unavoidable crashes, the authors employ a second stage check utilizing metamorphic testing [10], [11], [50].

In contrast to the approaches mentioned above, our method does not operate under an application-specific structure and hence is a generic fuzz testing methodology. So, instead of aiming to generate unique violations (e.g. based on a set of traffic rules), we attempt to generate a large number of failures where the failure type can be defined based on usage of the CPS. Finally, our coverage metric also differs from that of the aforementioned approaches in the sense that their metric is discrete (e.g. counting a discrete number of blocks traversed by the ego vehicle), while our approach considers continuous states of the CPS.

There are several other avenues proposed to generate test cases for Cyber-physical systems. To start, data-driven procedures have found uses in test case generation. In particular, Quindlen et al consider training an SVM to classify simulation trajectories as lying the "safe" or "unsafe" set. Furthermore, it actively samples new trajectories and ranks them to uncover the ones which will most improve the learning model [42].

Another approach leverages Rapidly-exploring Random Trees (RRTs) to generate test cases for CPS controllers. Bak et al uses RRTs to stress-test autonomous racing systems by modifying the behavior of other agents besides that of the ego vehicle. The large search space of these "adversarial agent perturbations" are explored using an RRT algorithm on a bounded projection of the simulation state space [8]. An earlier work also devised a modified RRT algorithm tailored towards generating test cases for motion-planning systems [29]. The algorithm proceeds under a new distance function and weighting scheme to account for system dynamics. Furthermore, the sampling probability distribution adapts to tree growth and exploration rate.

One of the widely applied techniques to discover bugs in CPS is Falsification [7][12]. When the safety specification of a CPS is provided as a formula in Signal Temporal Logic (STL) [13], falsification tools such as Breach [12] and S-TaLiRo [7][16] [17] employ stochastic optimization tools to generate an input to CPS that would violate the safety specification. There are two primary differences between fuzz-testing for CPS to generate failure test cases and falsification. First, is that falsification tools require the user to provide the specification in a formal logic. In comparison, fuzz-testing merely requires a failure model that specifies when is a run of CPS considered an error. Naturally, the failure model can include assertions over the software state as well as the state of the physical plant. Second, the goal of falsification is to generate an example trajectory that violates the safety specification. In contrast, the goal of fuzz testing is to generate a set of diverse inputs to CPS that cause the system to fail. The metric of success for a fuzz testing algorithm is not just the number of inputs that cause the system to fail, but also generate a coverage metric over the behaviors of CPS.

Reachability analysis has found usages in the verification of CPS. Specifically, reachability analysis can verify the safety of a system by attempting to compute the total region where a set of initial states can reach within a certain amount of time [6], [4], [5]. It has found further utility when combined with other techniques to perform falsification of hybrid systems and closed-loop control systems. [9], [24]. However, complex dynamics can make the exact computation of the reachable set untenable, and current methods may only output conservative over-approximations of the reachable set. These factors could decrease the amenability of reachablilty analysis for verification purposes.

3 BACKGROUND: COVERAGE-GUIDED FUZZ TESTING

Fuzz testing, also called fuzzing, is an automated testing technique that attempts to generate a diverse set of corner cases to ensure

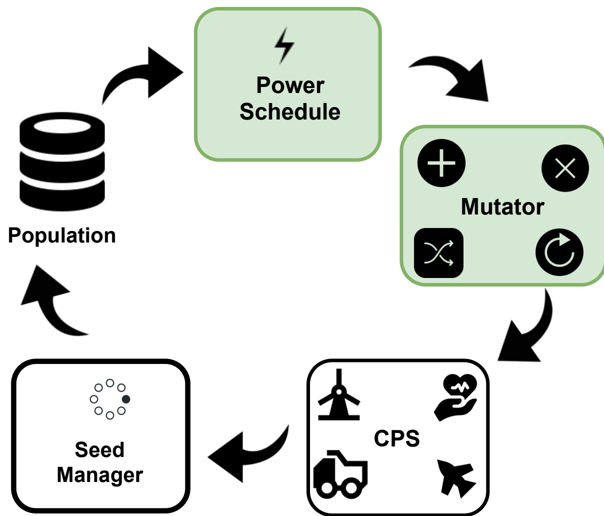


Figure 1: Fuzz testing architecture for CPS systems, where green boxes are our modifications.

robustness against exploitable vulnerabilities [39]. Targets may include files, network protocols, and software. The key idea behind fuzz testing is to stress-test the target program with automatically-generated test cases in order to trigger pathological behavior from the program. There exist various methods to produce suitable test cases and the most prevalent ones are coverage-guided strategies, genetic algorithms, symbolic execution, taint analysis, etc. These techniques allow the modern fuzz testing methodology to become an effective tool for revealing hidden bugs. So, as a unique testing approach whose success can be quantified in meaningful software-quality terms, fuzz testing plays an important theoretical and experimental role [32]. Moreover, fuzz testing has evolved to consider the static and dynamic information of an application [35]. In this section, we describe the coverage-guided fuzz testing.

Figure 1 presents an overview of coverage-guided fuzz testing. The process consists of several components:

- The target program is the *system-under-test (SUT)*, which takes inputs and produces outputs. It could be either executable or source code in different languages such as C++, Python, etc. As the source code of real-world software usually can not be accessed easily, so fuzz testing tools often target binary code. In our case, the system under test is a CPS simulator.
- **Seed**: Seeds are initial inputs to be mutated and changed to a specific degree of randomness to generate new inputs for SUT.
- **Population**: The set of all inputs, and the test results, which are fed as input into the power scheduler. We can think of the population as a container holding seeds with the seed manager being responsible for updating it. Initially, the inputs could be user-provided or randomly-selected.
- **Seed Manager**: The seed manager collects inputs and outputs from simulations of the CPS system and organizes the population. In our fuzz testing framework, it also keeps track of the frequency of usage of each seed. Then power schedule utilizes the frequency information as energy of the seeds.
- **Power Schedule**: Power schedule selects an input seed from the population for mutation. It is responsible for the fair distribution of simulation time such that promising seeds are more likely to be chosen for simulation. In the literature, the probability that a seed will be picked from the population is called the *seed energy*. The power schedule's task is to assign energy to seeds based on some predefined criteria, update seeds' energy, and finally prioritize seeds with higher energy for selection [48].
- **Mutator**: Although the essence of fuzz testing involves randomness, purely random test cases are often syntactically invalid and SUT throws them away at the early stages of execution. So, mutation-based fuzzing is utilized to generate functionally valuable input test cases.Mutation is the act of applying incremental changes to existing valid input seeds such that the modified seeds still remain valid but attributes new behavior. Mutation operations are often simple manipulations of the input seeds such as inserting a random character, omitting a character, flipping a bit, or shuffling a byte. The component carrying out mutation operations is called the *mutator* [48].

For fuzz testing CPS systems in this paper, we propose modifications to the power schedule as well as the mutator logic. They will operate with a new notion of coverage which we propose for CPS. These are described in more detail in the next section.

3.1 Fuzzing techniques

Broadly speaking, fuzzing techniques fall into one of three categories: white box, gray box, or black box. Membership depends on the amount of SUT internal information they require at run-time [27]. This information is used to guide test case generation and could include CPU utilization, memory consumption, data flow, or code coverage, etc.

- Black-box fuzz testers either mutate an input seed randomly or apply some conditions on a well-formed input seed. The mutation operations are typically bit flips, byte copies, or byte removals, etc [27]. The drawback of black-box fuzzing is generally low code coverage due to an uninformed mutation operation.
- Starting execution with a set of valid inputs, a white-box fuzz tester first gathers symbolic constraints at all conditional statements along the execution path under the inputs. After one execution, the white-box fuzz tester combines all symbolic constraints together using logic ANDs to form a path constraint (PC for short). Then, it systematically negates one of the constraints and solves the new PC. The new test case leads the program to run a different execution path. Using a coverage-maximizing heuristic search algorithm, white-box fuzz tester can find bugs in the target program as fast as possible [20].
  Theoretically, white-box fuzz testers are able to generate test cases for all execution paths and hence obtain full coverage. Practically, complex software design and the huge number

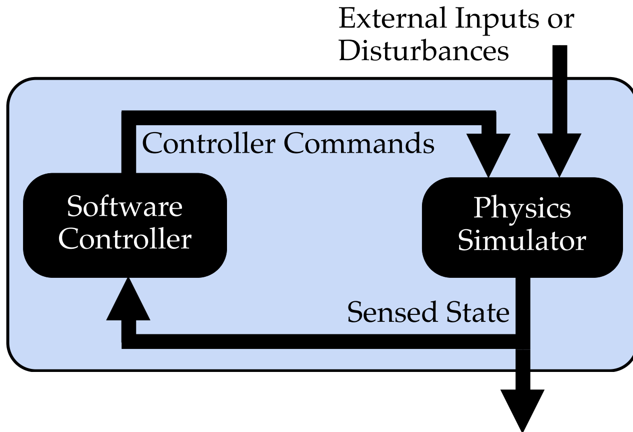


Figure 2: Our proposed coverage-guided fuzz tester for CPS works with black-box software and physics simulators, only using the sensed state output from the physics simulator.

of execution paths in addition to the imprecision of solving a symbolic execution constraint hinder the white-box fuzz testers to reach full code coverage.

- Gray-box fuzzing sits in the middle of white-box and black-box fuzzing. It works based on partial knowledge of the SUT commonly obtained through code instrumentation [15] [38]. Gray-box fuzz tester utilizes the code coverage information to adapt its mutation strategies to generate new test cases that possibly target more execution paths and trigger more software bugs.

3.2 CPS Execution Model

CPS systems pose a different set of challenges over generic software. An image of the system model we consider is shown in Figure 2. Rather than having a single input and output like a software function, systems will take inputs over time, corresponding to the environmental disturbances or external inputs. At each time step, the continuous state of the system will change and sensed input values will be put into a software controller that produces control commands. We do not assume we have access to the internals of either the software controller and physics simulator—they are black-boxes from the perspective of the test-generation mechanism. Application-specific error conditions are computed by the physics simulator and provided as part of the sensed state.

More formally, the physics simulator can be modeled as discrete-time function that advanced the internal simulation state by one step, $f : X \times U \times W \rightarrow Y$, with state $x \in X$, controller command $u \in U$, and external input or disturbance $w \in W$ and sensed output $y \in Y$. The software controller is a second function $g : Y \rightarrow U$, which operates on the observed state and produces a new controller command. In this work, we focus on the external input generation problem, so that we assume a fixed testing scenario (the initial state $x_0 \in X$ is given). The goal of fuzz testing is to find external input sequences, $w_0, w_1, \ldots w_T$, that cause the output y indicates some error condition, which we refer to as *interesting events* or simply *events*. For example, in our evaluation we will use an autonomous racing simulator as a CPS, where the physics simulation output includes a boolean flag indicating the presence of a collision. Furthermore, we would like to find external input sequences corresponding to events that cover the space of a CPS, which requires a custom notion of coverage for CPS that we define next.

4 FUZZ TESTING FOR CPS

In this section we present our framework for fuzz testing CPS that we call CPSFuzz.

4.1 Problem Statement

Definition 4.1. We model the Cyber-Physical System under test (SUT) as tuple $S = \langle X, T, I \rangle$ where

- X is the set of states. This set includes the software states X_s and the set of continuous states as X_c.
- I is the set of inputs to the system.
- $T : X \times I \rightarrow X$ is the transition function that defines the evolution of the system as a function of the current state and the input.

Definition 4.2. Given a SUT S and a failure specification F, the testing problem for a CPS is to generate a set of tests E that meet the failure specification.

Our goal is to generate test cases that make the system fail. We are trying to maximize the number of failing tests and increase the variety of failures.

4.2 CPS Coverage Metric

In order to perform coverage-guided fuzzing for CPS, we must define a notion of coverage. In software test generation, code coverage metrics such as branch or line coverage have been successfully used for this purpose. However, a CPS includes continuous states which are not captured by these traditional coverage metrics.

We next propose a metric that measures coverage when continuous states are present. Further, we will show how to integrate this metric score within the fuzz testing process in order to generate tests that attempt to maximize coverage. We assume the sensed state include information about *interesting events* or simply *events*, that can, for example, indicate unsafe configurations or system errors. In this framework we consider safety properties rather than history of temporal behaviors. The metric can be used to evaluate a testing strategy by providing a score to compare the amount of coverage for two testing approaches, even when the two approaches found a different number of interesting events each of which is in a different sensed state. The sensed state at these events are the input to the metric computation, which outputs a scalar coverage score, so that the metric score $\mathcal{S}$ takes in a finite set of sensed states of arbitrary size and outputs real-valued scalar, $\mathcal{S} : \text{Set}[Y] \rightarrow \mathbb{R}$. Besides operating on the sensed state of the system, we would like the metric to obey the following key properties:

(1) Adding more events never decreases the metric;
(2) Identical events do not increase the metric;
(3) Similar events have a lower impact on the metric than dissimilar events.

In order to achieve these goals, we build upon existing work in defining coverage for CPS [8]. Rather than considering the full

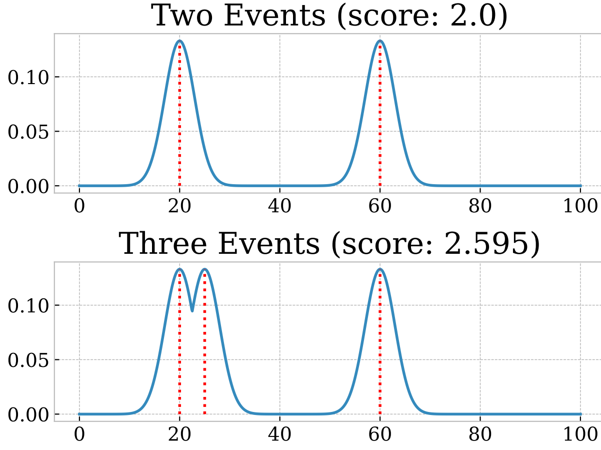


Figure 3: Our proposed CPS coverage metric increases as more interesting events are generated, although events close in the objective space do not increase the score as much as events that are separated.

sensed state Y, we focus the search on specific states based on the application. The user provides an **Objective Space Projection Function**: $\mathcal{P} : Y \rightarrow \mathbb{R}^o$ that maps sensed state to a o-dimensional Euclidean space called the *objective space* where we want to achieve coverage, as well as **Objective Space Exploration Limits**: $\mathcal{B} \in \mathbb{R}^{2o}$ that provide box bounds within the objective space.

Our metric is computed by taking each event, mapping it to the the objective spacing using $\mathcal{P}$, using a kernel function at the point that measures similarity of states, and then integrating the maximum of the kernels in the objective space over the box bounds $\mathcal{B}$. The kernel function is a hyper-parameter that measures the similarity of states in the objective space. We will use an o-dimensional normal distribution $\mathcal{N}(\mu, \sigma^2)$ for this purpose, where the mean μ is the point in the objective space of each interesting event and the standard deviation σ is a fixed hyper-parameter. The CPS coverage metric score $\mathcal{S}$ is computed by:

$$\mathcal{S}(\text{Set}[Y]) = \int_B \max_{y \in \text{Set}[Y]} \mathcal{N}(\mathcal{P}(y), \sigma^2)(b)\ \mathrm{d}b$$

An example of this score computation is shown in Figure 3. The objective space here is a single dimension along the x-axis, with limits $\mathcal{B} = (0, 100)$. At the top, two events (red dotted lines) are detected at 20 and 60 in the objective space. This is far apart with respect to the σ standard deviation hyper-parameter, which results in a score of close to two. At the bottom, a third event that maps to 25 in the objective space is added, which increases the score to about 2.595. Since this event was close to the other event at 20, the score does not increase to 3, as it would if we simply counted the number of interesting events. In the figure, the score (CPS coverage metric) in each case is computed by taking the integral of the blue curve from 0 to 100.

4.3 CPSFuzz Architecture

Fig. 1 depicts an overview of CPSFuzz architecture. This framework consists of several components:

- **Power schedule** is responsible for seed selection. It monitors the distribution of events happening at SUT's objective space and computes the CPS coverage score to deliver to the mutator. Also, it performs seed selection from the population based on the success rate of the seeds.
- **Mutator** performs various operations including but not limited to insertion, deletion, and update on the a valid seed based on the CPS coverage score.
- **SUT** is a CPS system for which our fuzz testing methodology generates inputs to cover its objective space.
- **Seed manager** is in charge of keeping the population updated by performing accounting operations on the seeds.
- **Population** is a repository containing valid seeds from successful simulation runs. It can be initialized by random or user provided valid seeds.

In the following, we will describe each component's mechanism in detail.

4.4 Power Schedule

```
  input  : Seed population: P, objective state space: Y
  output : valid seed: n, x ⊆ Y: X
1 energyList ← Energy(P);
2 n ← maximum{energyList} ;
3 for each y in Y do
4 |   score[y] ← S(y);
5 end
6 X ← state space(minimum (score));
7 return n, X;
```

Algorithm 1: Power Scheduler for determining the seed to be mutated and related objective state space.

As shown by Algorithm 1, CPSFuzz's power schedule proceeds through two phases: the first phase is seed selection from the population and the second phase is CPS coverage score computation for determining the mutation target. The energy level of a seed is inversely proportional to the frequency of the seed selection in previous test case generation runs. This strategy gives a fair chance to various valid seeds for selection hoping to capture new behavior upon mutation.

In software fuzz testing, a seed is directly sent to the mutator upon selection. However, CPS inputs are a sequence of control commands given to the plant, hence performing random mutation on the entire input sequence would not yield meaningful results. Therefore, to restrict the scope of the mutation, the power schedule finds a subset of the objective state space with minimum CPS coverage score. Our underlying observation is that the minimum CPS coverage score implies less interesting events and consequently a lack of behavior variety in that state space. So, by choosing the corresponding part of the input seed for mutation, those states would be the target of fuzz testing and exposed to the CPS system's new behavior.

4.5 Mutator

```
    input  : Seed: n, x ⊆ Y: X
    output : mutated seed: n′
1 start, end ← map_states(X);
2 while condition do
3 |   operator ← selectAtRandom(operators) ;
4 |   n′ ← mutate(n, start, end, operator);
5 end
6 return n′;
```

Algorithm 2: Mutator that performs mutations over the seed to generate new test inputs.

Based on the Algorithm 2, the mutator takes the valid seed n and a subset of objective state space X determined by the power schedule as input and maps X to a corresponding interval in the input sequence. Then, it modifies the interval in the seed using one of the mutation operators chosen at random, leaving other parts of the seed unchanged making sure that the possible new behavior originates from the modified interval.

Software fuzz testing methods leverage bit and byte level manipulation on inputs. However, fine-grained operations are not suitable for CPS control sequences as the goal is not to obtain code coverage but causing state changes.So, CPSFuzz employs coarse-grained mutation operations for CPS inputs, including:

- Insert one or multiple random control command(s) to the seed at a random position(s) in the interval $[\text{start}, \text{end}]$.
- Remove one or multiple random command(s) from the seed at interval $[\text{start}, \text{end}]$.
- Update one or multiple random command(s) with random control inputs in the seed at interval $[\text{start}, \text{end}]$.

4.6 Seed manager

Seed manager is responsible for adding new valid seeds to the population or removing seeds. At the beginning, the population is initialized with some valid random inputs. After each execution, if SUT does not fail on the mutated seed, the seed manager inserts the mutated input as a valid seed to the population. Seed manager periodically eliminates the low energy seeds, as their low energy reflects their selection history. Also, the seed manager updates the seeds' energy whenever they are selected for mutation.

5 EVALUATION

To evaluate our work, we conducted a set of experiments using CPSFuzz and three other fuzz testing methods on an autonomous racing project called F1tenth [2]. Complete source code of CPSFuzz and the experiments' settings and configurations are available at [44]. In the following sections, we introduce the F1tenth project, briefly explain the fuzz testing approaches and present the results. Our goal is generation of test case inputs which induce the cars to either collide with each other or the track boundaries.

5.1 F1Tenth

The F1Tenth gym environment is an open-source autonomous racing simulation platform capable of hosting several vehicle instances modeled with realistic dynamics. Its deterministic structure gives the opportunity to reproduce the results. Also, the simulations decrease the run-time up to 30 times in comparison to real-time execution [40].

CPSFuzz generates test case inputs for the controller of the cars by changing the vehicles' velocity to induce crashes into the track boundaries and other cars. These collisions are automatically detected by the gym environment, and CPSFuzz considers them as failures.

5.2 Fuzz Testing Methodologies

For evaluation, we repeated the experiments with the following three fuzz testing approaches, and compared the quality of their generated test cases with CPSFuzz's.

- **Atheris**: Atheris is an open-source coverage-guided fuzz testing tool for Python programs [1]. However, as it is based on libFuzzer, a fuzz testing library for the C/C++ languages, it can support native extensions written for CPython.
- **Hypothesis**: Hypothesis performs property-based testing [34]. It defines a statement that is true based on the code "properties", and then generates test inputs (typically randomly generated inputs of an appropriate type), and observes whether execution with the input violates any of the properties. Property violation generally means capturing a bug or error in the code.
- **Random Fuzzing method**: We developed a pure random fuzz testing methodology as our baseline. It is a black-box method that generates inputs purely at random while taking no metric or property into consideration.

It is worth mentioning AFL (American Fuzzy Lop) [47] which is a grey-box fuzz testing framework. It leverages code coverage information to figure out how to target different parts of the program. AFL uses program instrumentation to extract branch coverage information of an input. For each input AFL knows the frequency at which each branch is exercised. If an input contributes to new coverage it will be used for further mutation and new input generation. [48]. We conducted a set of experiments using AFL and surprisingly observed that it was not able to produce a reasonable amount of test cases. The main reason is that it is bound to the source code instrumentation, while accessibility to source code is the main issue with the instrumentation based approaches. Specifically, CPS are generally black-box and AFL fails to instrument them. To assure this is the main reason, we carried out a set of experiments seeking to instrument a CPS and perform code coverage-guided fuzz testing. As expected, we obtained the same unsatisfactory result as a confirmation to AFL functionality.

Table 1 shows the results of test case generation by the four fuzz testing methodologies during five runs of each taking 1M time frames, where each 100 frames is equal to 1 second. We used this logical execution time to eliminate the effect of other factors such as hardware speed on run-time.

The second column shows the total number of input test cases generated by each method. Test cases include those that cause the cars to crash (invalid inputs) and those allowing the cars finish the lap without any collision (valid inputs). The third column states the rate of the invalid input generation. The fourth column reflects

Fuzzer	# Test cases	# Invalid Test cases	Score
CPSFuzz	361	47.48%	21.06
Atheris	635	44.09%	3.28
Hypothesis	562	89.50%	13.54
Random	499	84.36%	16.34

Table 1: Median scores during five runs of test case generaion, one million frames at each run

the CPS coverage metric score of each approach for test case generation. CPS coverage metric does not account the rate of test case generation but rather the rate of unique invalid test case generation. In practice, all four approaches generate a huge number of invalid inputs that cuased the cars crashed, however, most of them are identical and cause the cars crash around the same area such as specific turns. Meaning, a large subset of the objective state space is left untouched. In the F1Tenth project, we considered the objective state space to be the race track and set our metric to be variety of crashes covering the track. So, the CPS coverage metric scores convey the amount of coverage the generated test cases expand. In other words, the approaches which don't generate various unique test case inputs will lose most of the CPS coverage metric score.

As the number of test cases shows, CPSFuzz generates fewer test cases during a fixed time window than the other three methodologies and consequently, its invalid test cases are less than them, but it achieves the highest CPS coverage metric score among all the other methods revealing that it spends the fuzz testing time on generation of unique invalid inputs. In contrast, although the other methods generate more invalid test cases but they get lower scores as their generated tests lack variety and are mostly identical.

5.3 Comparison of CPS Coverage Metric

In our proposed metric from Section 4.2, we computed the coverage score using $\mathcal{S}$, which integrated the maximum of a set of Gaussian distributions in the objective space. We first compare this metric to an existing work on test-generation for autonomous systems [8], where the metric for measuring uniqueness of failures used the spatial clustering DBScan [14] algorithm. This approach distinguishes the uniqueness of failures by coalescing similar events into clusters, so that failures near the same location do not contribute to the number of unique events. Two scenario-specific hyper-parameters are needed to determine when points are considered to fall in the same cluster. For comparison, we use the same parameters as the earlier work. The maximum distance between two samples to be considered in the same cluster is 2.1 meters, and the number of samples required to be in the neighborhood to consider it as cluster is set to 3. The number of clusters plus outliers, which fall outside of any cluster, represents the score computed as unique failures.

We applied this metric and compared our methodology using the other three fuzz testing methods. DBScan results reveal that our method has a higher rate of unique failure generation than the other three methods where CPSFuzz generated 6 clusters and six outliers, Random approach produced 4 clusters, Atheris provided one cluster and Hypothesis ended up in 5 clusters and 3 outliers. Figure 4 is a demonstration of DBScan metric revealing that CPSFuzz is providing higher coverage in comparison to Random approach.

Although the earlier DBScan metric does judge our fuzz testing methodology, it is not an ideal metric. Specifically, adding more points can reduce the score, as different clusters may be merged together and considered a single unique failure (more events can lower the score). This is apparent in the figure, where a large number of crashes on the left side of the track are merged together into a single large cluster.

Figure 5, in contrast, shows the coverage score computed using our CPS coverage metric , as more simulation steps are performed. For our metric, we used a single-dimensional objective space by converting the car's x and y position output by the physics simulator into a percent track completed position, with limits at 0 and 100 (this is a normalized version of the s value of the car position in the Front frame). In this set of experiments we chose the standard deviation to be 1 ($\sigma = 1$).

We compare four approaches: Hypothesis, random testing, Atheris, and CPSFuzz. Hypothesis is a property-based fuzz testing tool. Here, we defined the property to be the lack of collisions and Hypothesis aimed to generate test case inputs designed to violate this property. However, coverage in terms of track position was not taken into account.

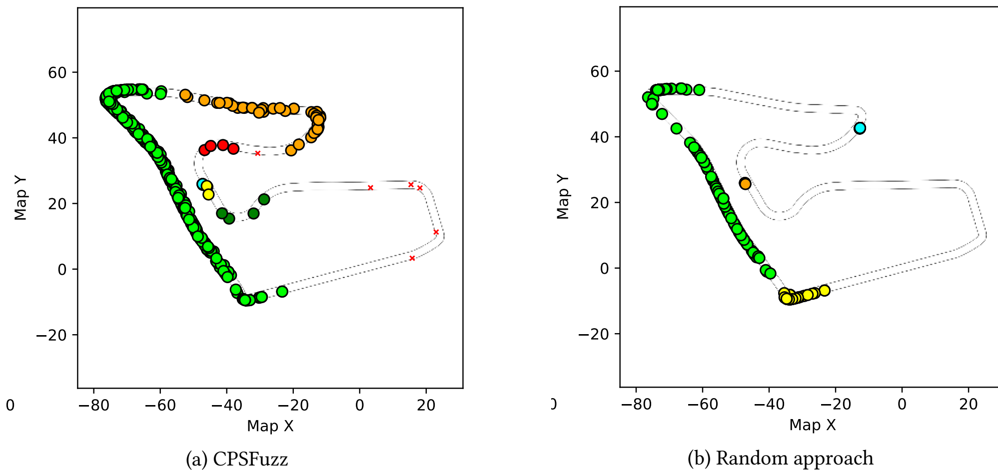
The random approach is a black-box method that generates all its test case inputs using random external inputs. Atheris is a coverage-guided tool and performs bit-level mutation operations on input seeds to generate new test cases hitting various branches and code segments. CPSFuzz is our proposed method, which mutates the input seeds with the goal of improving the CPS coverage metric as described in Section 4.

We carried out several sets of experiments using all four approaches and, due to randomness, we report the median results in Figure 5. CPSFuzz did 1.5x better than Hypothesis, 1.3x better than random approach, 6.4x better than Atheris.

Figure 6 shows the score function $\mathcal{S}$ that is integrated to compute CPS coverage metric . The diagram in Figure 6.a shows that CPSFuzz obtained the maximum score for most of the segments of the race track. This is indicative of the failure-inducing test cases containing unique collisions on most parts of the race track. Figure 6.b and 6.d show that Hypothesis and the random approach generated unique failure test case inputs for about the first 40% of the track but were unable to properly extend the coverage beyond, except in a few cases. Fig 6.c shows that Atheris only finds crashes on the first 15% of the track. One of the main reasons for this poor performance is the kind of fine-grained mutation operations that Atheris performs, such as shuffle bytes or shift bits that do not modify the input sequences meaningfully in a CPS Scenario where code coverage is not a metric. CPSFuzz, on the other hand, makes coarse-grained modification to input seeds, in a higher level abstraction close to controller's commands, targeting the uncovered continuous states in the system.

6 DISCUSSION

One result of our study is the demonstration of generating a large number of different failure cases using a fuzz testing methodology



(a) CPSFuzz

(b) Random approach

Figure 4: Although CPSFuzz found more unique crashes using the DBScan metric, crashes along the left side of the track were combined into a single cluster (green), showing the shortcoming of the earlier metric.

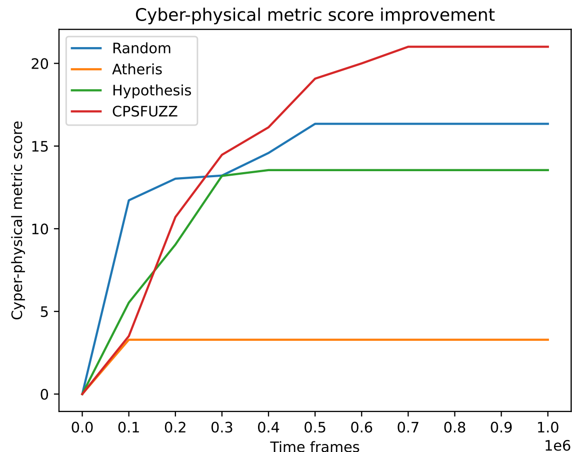


Figure 5: Comparing CPS coverage metric improvement through time for the four fuzz testing methodologies

for CPS systems. Classical testing approaches or manually generating such tests would be infeasible. Moreover, this variety of test scenarios opens up other types of investigations. For instance, for the autonomous racing case study, we could analyze the state of the system at a crash to detect additional information to differentiate crashes. This could be used to identify who was at fault, whether the opponent car hit the ego car at the back left, center, or right and so on, to further create varying test cases.

From the test strategy and configuration perspective, there exist many parameter tuning and configurations which could affect the test results. For the CPS coverage metric we tried various standard deviation (SD) values and observed that it alters the CPS coverage metric score. However, it affects the absolute score values while leaving the relative scores of the approaches unchanged.

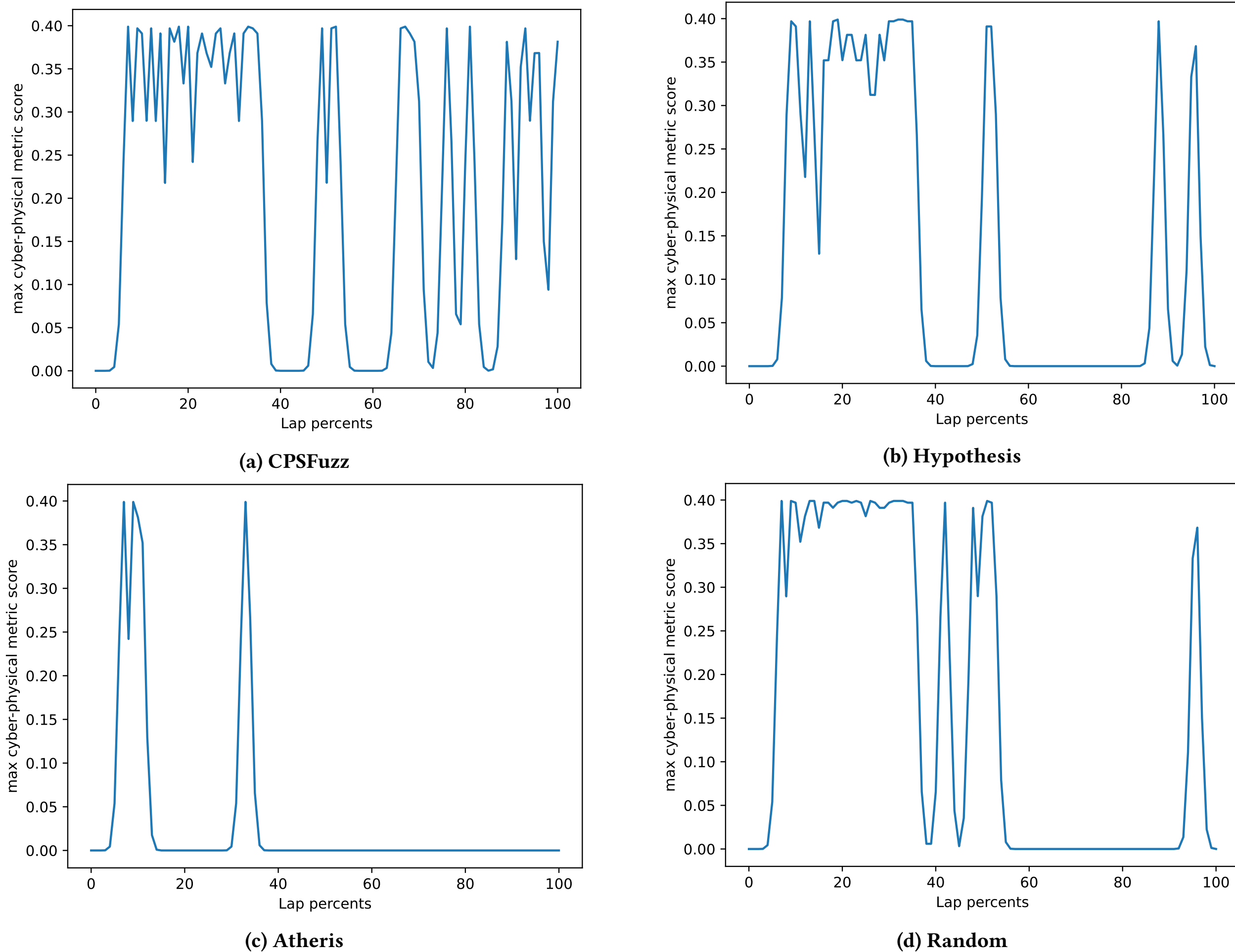
Another possible venue for exploration would be the integration of application-specific contextual information to enrich test scenarios or improve efficiency. As an example, we set the testing framework to terminate the simulation runs that were not close to collision generation using the information obtained from the agents and the environment and drastically decreased the simulation time as the long runs where cars successfully cross the finish line have no testing value but increasing the simulation time.

7 CONCLUSION

Cyber-physical systems have plenty of safety-critical applications and must go under heavy stress tests. In this work, we proposed a modified fuzz testing methodology, CPSFuzz, for testing CPS systems. We introduced a new metric for test coverage while dealing with black-box CPS simulators. Moreover, we utilized this CPS coverage metric in designing customized power schedule and mutator modules, both of which are major components of coverage-based fuzz testing, to boost test case diversity and hence improve the coverage score. We evaluated our approach using the F1Tenth autonomous racing project and compared its functionality to some other fuzz testing methods known in the field. Results demonstrate that CPSFuzz functionality was 1.5x better than Hypothesis, 1.3x better than random approach, 6.4x better than Atheris.

8 ACKNOWLEDGMENTS

This material is based upon work supported by the Air Force Office of Scientific Research and the Office of Naval Research under award number FA9550-19-1-0288, FA9550-21-1-0121, FA9550-22-1-0450 and N00014-22-1-2156. Also, we would like to acknowledge the support of National Science Foundation (NSF) under grant numbers CNS 2038960, and Amazon Research Award — Automated Reasoning. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force, the United States Navy, National Science Foundation, or Amazon.



(a) CPSFuzz

(b) Hypothesis

(c) Atheris

(d) Random

Figure 6: Maximum CPS coverage score obtained by the four fuzz testing methodologies for autonomous racing, where the x axis is projection of state space to one-dimentional objective space, percent track completed position.

REFERENCES

[1] Atheris: A coverage-guided, native python fuzzer. https://github.com/google/atheris.

[2] F1tenth. https://f1tenth.org/.

[3] libfuzzer –– a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2020.

[4] Matthias Althoff. *Reachability analysis and its application to the safety assessment of autonomous cars*. PhD thesis, Technische Universität München, 2010.

[5] Matthias Althoff and John M Dolan. Online verification of automated road vehicles using reachability analysis. *IEEE Transactions on Robotics*, 30(4):903–918, 2014.

[6] Matthias Althoff, Olaf Stursberg, and Martin Buss. Verification of uncertain embedded systems by computing reachable sets based on zonotopes. *IFAC Proceedings Volumes*, 41(2):5125–5130, 2008.

[7] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. S-taliro: A tool for temporal logic falsification for hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 254–257. Springer, 2011.

[8] Stanley Bak, Johannes Betz, Abhinav Chawla, Hongrui Zheng, and Rahul Mangharam. Stress testing autonomous racing overtake maneuvers with rrt. *arXiv preprint arXiv:2110.01095*, 2021.

[9] Sergiy Bogomolov, Goran Frehse, Amit Gurung, Dongxu Li, Georg Martius, and Rajarshi Ray. Falsification of hybrid systems using symbolic reachability and trajectory splicing. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pages 1–10, 2019.

[10] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)*, 51(1):1–27, 2018.

[11] Tsong Yueh Chen, Tsun Him Tse, and Z. Quan Zhou. Fault-based testing without the need of oracles. *Information and Software Technology*, 45(1):1–9, 2003.

[12] Alexandre Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *International Conference on Computer Aided Verification*, pages 167–170. Springer, 2010.

[13] Alexandre Donzé and Oded Maler. Robust satisfaction of temporal logic over real-valued signals. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 92–106. Springer, 2010.

[14] M. Ester, J. Sander H.-P. Kriegel, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *in Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, page 89–100, 1996.

[15] Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 190–200, 2005.

[16] Georgios E. Fainekos and George J. Pappas. Robustness of temporal logic specifications for continuous-time signals. *Theoretical Computer Science*, 410(42):4262–4291, 2009.

[17] Georgios E Fainekos and George J Pappas. Robustness of temporal logic specifications for continuous-time signals. *Theoretical Computer Science*, 410(42):4262–4291, 2009.

- [18] Seyed K Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. {BUZZ}: Testing context-dependent policies in stateful networks. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 275–289, 2016.
- [19] Daniel S Fowler, Jeremy Bryans, Siraj Ahmed Shaikh, and Paul Wooderson. Fuzz testing for automotive cyber-security. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 239–246. IEEE, 2018.
- [20] P. Godefroid and D. A. Molnar M. Y. Levin. Automated whitebox fuzz testing. In *in Proc. Netw. Distrib. Syst. Security Symp.*, volume 14, pages 1–16, 2008.
- [21] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20–27, 2012.
- [22] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.
- [23] Brandon Wen Heng Goh. American fuzzy lop (afl) fuzzing. 2019.
- [24] Manish Goyal and Parasara Sridhar Duggirala. Neuralexplorer: state space exploration of closed loop control systems using neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 75–91. Springer, 2020.
- [25] Jia Cheng Han and Zhi Quan Zhou. Metamorphic fuzz testing of autonomous vehicles. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 380–385, 2020.
- [26] Zhisheng Hu, Shengjian Guo, Zhenyu Zhong, and Kang Li. Coverage-based scene fuzzing for virtual autonomous driving testing. *arXiv preprint arXiv:2106.00873*, 2021.
- [27] Esa Jääskelä. Genetic algorithm in code coverage guided fuzz testing. 2016.
- [28] Dave Jones, T Rantala, and V Weaver. Trinity: A linux system call fuzz tester, 2016.
- [29] Jongwoo Kim, Joel M Esposito, and Vijay Kumar. Sampling-based algorithm for testing and validating robot controllers. *The International Journal of Robotics Research*, 25(12):1257–1272, 2006.
- [30] Ralph LaBarge and Thomas McGuire. Cloud penetration testing. *arXiv preprint arXiv:1301.1912*, 2013.
- [31] Guanpeng Li, Yiran Li, Saurabh Jha, Timothy Tsai, Michael Sullivan, Siva Kumar Sastry Hari, Zbigniew Kalbarczyk, and Ravishankar Iyer. Av-fuzzer: Finding safety violations in autonomous driving systems. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 25–36. IEEE, 2020.
- [32] H. Liang, X. Jia X. Pei, and J. Zhang W. Shen. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, Sep 2018.
- [33] 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.
- [34] David R. MacIver and Zac Hatfield-Dodds. Hypothesis. https://github.com/HypothesisWorks/hypothesis.
- [35] G. McGraw. Silver bullet talks with bart miller. *IEEE Security Privacy*, 12(5):6–8, Sep 2014.
- [36] 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.
- [37] Lama J Moukahal, Mohammad Zulkernine, and Martin Soukup. Vulnerability-oriented fuzz testing for connected autonomous vehicle systems. *IEEE Transactions on Reliability*, 2021.
- [38] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 89–100, 2007.
- [39] P. Oehlert. Violating assumptions with fuzzing. In *IEEE Security Privacy*, volume 3, pages 58–62, 2005.
- [40] M. O'Kelly, D. Karthik H. Zheng, and R. Mangharam. F1tenth: An open-source evaluation environment for continuous control and reinforcement learning. In *in Proceedings of the NeurIPS 2019 Competition and Demonstration Track, ser. Proceedings of Machine Learning Research, H. J. Escalante and R. Hadsell, Eds.*, page 77–89, 2020.
- [41] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318. PMLR, 2013.
- [42] John F Quindlen, Ufuk Topcu, Girish Chowdhary, and Jonathan P How. Active sampling-based binary verification of dynamical systems. In *2018 AIAA Guidance, Navigation, and Control Conference*, page 1107, 2018.
- [43] Dimitrios Serpanos and Konstantinos Katsigiannis. Fuzzing: Cyberphysical system testing for security and dependability. *Computer*, 54(9):86–89, 2021.
- [44] Sanaz Sheikhi. Cpsfuzz project repository. https://github.com/sanazsheikhi/CPSFuzz/tree/master.
- [45] Dmitry Vyukov. Syzkaller—linux kernel fuzzer.
- [46] Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating smt solvers via semantic fusion. In *Proceedings of the 2020 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*, 2020.
- [47] Michal Zalewski. American fuzzy lop. https://lcamtuf.coredump.cx/afl/, 2014.
- [48] A Zeller, R Gopinath, M Böhme, G Fraser, and C Holler. The fuzzing book. *The Fuzzing Book. Saarland University*, 2019.
- [49] Ziyuan Zhong, Gail Kaiser, and Baishakhi Ray. Neural network guided evolutionary fuzzing for finding traffic violations of autonomous vehicles. *arXiv preprint arXiv:2109.06126*, 2021.
- [50] Zhi Quan Zhou and Liqun Sun. Metamorphic testing of driverless cars. *Communications of the ACM*, 62(3):61–67, 2019.