IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 27, 2020, accepted September 8, 2020, date of publication September 10, 2020,
date of current version September 23, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3023250

CPFuzz: Combining Fuzzing and Falsification of Cyber-Physical Systems

FUTE SHANG[1], BUHONG WANG[1], TENGYAO LI[1], JIWEI TIAN[1], AND KUNRUI CAO[1,2]
[1]School of Information and Navigation, Air Force Engineering University, Xi'an 710089, China
[2]School of Information and Communications, National University of Defense Technology, Xi'an 102249, China

Corresponding author: Buhong Wang (wbhgroup@aliyun.com)

ABSTRACT Coverage-guided grey-box fuzzing for computer systems has been explored for decades. However, existing techniques do not adequately explore the space of continuous behaviors in Cyber-Physical Systems (CPSs), which may miss safety-critical bugs. Optimization-guided falsification is promising to find violations of safety specifications, but not suitable for identifying traditional program bugs. This article presents a fuzzing process for finding safety violations at the development phase, which is guided by two quantities: a branch coverage metric to explore discrete program behaviors and a Linear Temporal Logic (LTL) robust satisfaction metric to identify undesirable continuous plant behaviors. We implement CPFuzz to demonstrate the utility of the idea and estimate its effectiveness on seven control system benchmarks. The results show up to a better performance in average time to find violations on all benchmarks than S-TaLiRo and six benchmarks than S3CAMX. Finally, we exploit CPFuzz to synthesize the sensor spoofing attack on a DC motor with fixed-point overflow vulnerability as a case study.

INDEX TERMS Coverage guided fuzzing, cyber-physical systems, linear temporal logic, optimization-guided falsification.

I. INTRODUCTION

The problem of falsifying a safety property for CPS has extensively been studied during the last years. Optimization-guided falsification simulates the system on intelligently generated inputs and feeds back the corresponding traces to find system violations more effectively. The robust satisfaction semantics of temporal logic [1] map the trace of the system executing to a real value instead of a logic value, offering more gradient information for optimization. Based on the robust satisfaction semantics of temporal logic, falsification casts the problem of searching safety violations as an optimization problem. The CPS initial state and the parameterized input signal are used as decision variables. The robust satisfaction semantics of temporal logic are used as the cost function for the optimization problem, which is highly nonlinear and discontinuous. Therefore, optimization-guided falsification adopts a variety of heuristic optimization algorithms, such as ant colony algorithm [2], simulated annealing [3], reinforcement learning [4] and so on.

This paper focuses on two problems in optimization-guided falsification. In many industrial-scale CPSs with complex controller (modern cars may contain more than 10^8 lines of code), the cost function may lack gradient information, making it difficult for the heuristic optimization algorithm to find the search direction. Experiments in [5] show that this situation often occurs when the controller code has nested logical conditions about continuous variables, or when there are discrete variables that indicate the operating mode. The cost function can hardly guide the construction of new test cases, degenerating from an optimization problem to a random search in the problem space. To improve the search efficiency, we introduce more feedback information about the controller execution.

On the other hand, most of the CPS analysis methods [6] focus on theoretical models, for example, hybrid automata. These analysis methods could find errors in the design phase. However, they abstract away the specific implementation details and ignore potential bugs, such as logic errors in implementation [5], fixed-point overflow [7], integer overflows [8], etc. These bugs may cause the controller to fail to work or cause safety problems. Therefore, the controller binary vulnerability analysis is necessary to identify bugs introduced at the development phase of CPS.

Symbolic execution techniques [8], [9] have been adopted to analyze the controller, which inspires us to use other

The associate editor coordinating the review of this manuscript and approving it for publication was Tiago Cruz.

program analysis methods to solve the problem in CPS, explicitly speaking, fuzzing. Fuzzing is a testing method to generate random input to trigger crashes in the program. Each crash may be caused by a specific error in the program, like memory overflow. If the input satisfies the condition to trigger the crash, it is valuable to help programmers find the bugs in the program. Feedback is also used in fuzzing. If a test case causes the code to branch a different way at an if-statement, it is more likely to generate other inputs that cover new execution paths.

A higher path coverage represents a greater possibility of triggering crashes embedded in the program's deep logic. Based on this observation, coverage guided fuzzing employs compile-time or run-time instrumentation technique to get feedback information of branch coverage. The test case with higher coverage is allowed to generate more test cases by mutating this one. Falsification and fuzzing are both feedback based testing methods, and there are many similarities between them. Could we solve the above research problems by combining fuzzing and falsification?

The goal of this paper is to identify safety violations of CPS in the development phase efficiently. We hope to integrate robust satisfaction semantics of temporal logic in coverage-guided fuzzing and obtain a cyber-physical fuzzing framework. In our work, CPS is considered a loop of the interwoven controller and physical plant at a fixed rate. In each iteration, the sensor values $\mathbf{y}$ are sampled periodically and then perturbed by the disturbance input $\Delta\mathbf{y}$ according to perturbation function. The domain of $\Delta\mathbf{y}$ is also user-specific. The controller reads contaminated inputs and outputs the result $\mathbf{u}$, which is held constant before the controller calculates a new output. The controller is instrumented to record the program path information at every branch. The physical dynamics model of the plant is specified as a black-box function $\mathbf{SIM}(\mathbf{x}, \mathbf{u}, i)$ that simulates the dynamics starting from a current state $\mathbf{x}_i$ and under an input $\mathbf{u}_i$ of the i-th sampling period. Under the assumption that the trace of the plant $\mathbf{x}$ is fully observable, we can obtain a robust satisfaction value. Given the branch coverage from the controller and robust satisfaction value from the plant, we could calculate the energy for current input. Energy represents how many offsprings are allowed to generate for one input, which is used to prioritize inputs covering more paths or leading systems to an unsafe state. We propose three mutation operators to generate valid inputs, including the CPS initial states and the adversarial perturbations for the sensor.

In our research, we have implemented a full-featured prototype of CPFuzz based on American Fuzzy Lop (AFL) [10], a top-rated coverage guided fuzzing tool. To evaluate its effectiveness, we run our implementation on seven benchmarks from [5], [11]–[13]. We evaluate configurations of mutation operators to increase the effectiveness of CPFuzz. We also compare CPFuzz with mature tools, S-TaLiRo [14], S3CAMX [5], and an adapted version of AFL. S3CAMX uses static symbolic execution on the controller program to find violations. The adapted version of AFL is the same as CPFuzz except that it does not exploit robustness to guide the search. CPFuzz demonstrates the performance improvements, especially in the ability to find a violation in the controller with more paths. CPFuzz could be used to identify more violations that are undetectable by traditional fuzzing and falsification tools. We demonstrates the ability in a case study by the synthesis of the sensor spoofing attack on a DC motor with fixed-point overflow vulnerability.

The contributions of this work can be summarized as follows:

1) We propose the cyber-physical fuzzing process. To the best of our knowledge, it is the first general feedback-based fuzzing technique guided by a combination of two metrics: a metric quantifying program branch coverage and a metric quantifying the robust satisfaction value of the given specification.
2) We develop CPFuzz, a proof-of-concept prototype to demonstrate the feasibility of automatically identifying violations in CPS implementation.
3) We compare the performance of our implementation with S-TaLiRo and S3CAM-X over seven benchmarks. The results show up to a better performance in average time for 10 runs to find violations on all benchmarks than S-TaLiRo and six benchmarks than S3CAMX. As shown in the DC motor example, CPFuzz could exploit implementation vulnerabilities to violate the safe specifications.

The remainder of this work is organized in the following manner. Section II and Section III describe the formalized problem and proposed approach in this research, respectively. Section IV and Section V present the implementation and detailed analysis of the experiments. We conclude the paper in Section VI.

A. RELATED WORKS

Our technique brings together two lines of work. The first is the falsifying temporal specifications on CPS models. The second line of work is mutation-based fuzzing.

Optimization-guided falsification is an emerging approach to test CPS for undesirable model behaviors guided by the safety specification [6]. Coverage-guided falsification could not only find violations faster but also increase the number of founded violations to provide a more reliable correctness guarantee. Various notions are proposed to measure the coverage [15]–[17]. None of this line of work focused on the structure coverage of the controller software, which could also guide the exploration of the solution space and trigger program logic to output unsafe control command. Compared to the prior work on coverage-guided falsification, CPFuzz takes a step further and explores the state space of a CPS guided by the controller's branch coverage to improve the efficiency of falsification.

Mutation-based fuzzing is a practical approach to find vulnerabilities in traditional software, so it is natural to

ask whether it could be applied to CPS. There are some studies on fuzzing CPS. Kim *et al.* [18] found the input validation bugs in robotic vehicles control programs and used fuzzing to search the controller parameters. Chen *et al.* [19] proposed to use a genetic algorithm to guide the fuzzing of actuators to drive the CPS into different unsafe physical states. This paper searches the actuator configuration in a bit vector under the assumption that the states of actuators are discrete. In that vein, their following research [20] used online active learning to guide a search for network packet payload, which encodes actuator commands to drive the CPS into an unsafe state. In our paper, there are a couple of differences to notice: none of this line of work exploited the program coverage information, which may provide useful information for optimization, while our approach is based on coverage guided fuzzing technique, which has many success stories in large-scale software [21]–[23]. Then, they focused on avoiding an unsafe set; this restriction allows search heuristics that rely on spacial metrics. In our current work, we allow arbitrary LTL specifications and use robustness satisfaction values as guidance.

II. BACKGROUND

A. FORMAL CPS MODEL

Our approach considers a simplified model of the CPS that is composed of two parts, the program implementation of the controller and the physical dynamics black-box model. The models are adapted from the research [5].

Definition 1 (Physical Dynamics Model): The physical dynamics model is described by a set of physical states $\mathcal{X}$, model inputs $\mathcal{U}$, outputs $\mathcal{Y}$ and the total simulation time horizon T along with two functions:

1) A simulation function $\mathbf{SIM} : \mathcal{X} \times \mathcal{U} \times \mathbb{R}_{\geq 0} \to \mathcal{X}$, where $\mathbf{SIM}(\mathbf{x}_i, \mathbf{u}_i, \Delta T)$ maps the current state $\mathbf{x}_i$ at the ith sampling time to the next state $\mathbf{x}_{i+1}$ at the $i+1$th sampling time (after time step $\Delta T \geq 0$) with the assumption that the input signal $u(t)$ is a constant $\mathbf{u}_i \in \mathcal{U}$ for $t \in [0, \Delta T)$. The simulation ends at time T.
2) An observation function $g : \mathcal{X} \to \mathcal{Y}$ that maps the current state $\mathbf{x}_i$ to the observable output $\mathbf{y}_i = g(\mathbf{x}_i)$.

The physical dynamics take the controller output $\mathbf{u}_i$ at the ith sampling period as input and output the sensor value $\mathbf{y}_i$ of system state. The simulation function **SIM** can be a nonlinear hybrid system or even a data-driven model, such as a neural network that maps a current state to a next state. The initial state $\mathbf{x}_0$ of the CPS is generated by the fuzzer to explore the possible configurations of the physical environment.

Definition 2 (Controller Model): A controller is specified in terms of its input space $\mathcal{Y}$, its internal state space $\mathcal{S}$, and the controller sampling period ΔT. Its semantics are provided by a function $\rho : \mathcal{Y} \times \mathcal{S} \to \mathcal{U} \times \mathcal{S}$, where the function $\rho(\mathbf{y}_i, \mathbf{s}_i)$ maps the controller input $\mathbf{y}_i$ and internal state $\mathbf{s}_i$ to $(\mathbf{s}_{i+1}, \mathbf{u}_i)$, where $\mathbf{s}_{i+1}$ and $\mathbf{u}_i$ are the updated controller state and the input to the plant after time step ΔT, respectively.

The controller gets the input $\mathbf{y}_i$ at the sampling period i repeatedly and outputs $\mathbf{u}_i$. We assume the controller will finish the computation in a sampling period without time delay and hold the output value in the sampling period i.

Definition 3 (Execution Trace): Given a sampling period ΔT, the operational semantics of the CPS model can be described as a execution trace $\mathbf{x}$ defined as a sequence of states at time $i \cdot \Delta T$:

$$\mathbf{x} = (\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_N) \tag{1}$$

where $N = T/\Delta T$.

B. THREAT MODEL

This article assumes a spoofing attack model. An attacker can spoof the original sensor value $\mathbf{y}_i$ with an attack signal $\Delta\mathbf{y}_i$. To enhance the rationality, the perturbation $\Delta\mathbf{y}_i$ has a certain range of values $\mathcal{Y}'$ given by users. If it is too small, the injected disturbance may be overwhelmed by the noise. If it is too large, it may exceed the sensor's output range and be directly filtered by the anomaly detection algorithm in the controller. This range needs to be configured according to the characteristics of the sensor. The attack model defines the function $\hat{\mathbf{y}}_i = pert(\mathbf{y}_i, \Delta\mathbf{y}_i)$ that combines the attack signal with normal sensor signal. In the false data injection attack of the power grid, the attack signal can be directly added to the normal sensor signal $\hat{\mathbf{y}}_i = \mathbf{y}_i + \Delta\mathbf{y}_i$.

The attack aims to ultimately affect the physical system, causing the system to deviate from the original control target and enter an unsafe state set by the attacker, such as bypassing anomaly detection, leading to physical collision or usability issues. Traditional fuzzing methods could not detect this kind of problem as controller safety-critical bugs may not lead to a program crash, a standard indicator of traditional bugs (e.g., memory corruption). We adopt the Linear Temporal Logic (LTL) [24] to describe the unsafe state of the system, which has the following core grammar:

$$\varphi ::= \top | \mu | \neg\varphi | \varphi \wedge \varphi | \varphi \mathbf{U}_{(0,T)} \varphi \tag{2}$$

Below is an example of the LTL specification to describe the position (x, y) of an aircraft can never be in the square $((x_l, y_l), (x_r, y_r))$ within the time horizon T:

$$\phi_d = \Box_{(0,T)} \neg(x > x_l \wedge x < x_r \wedge y > y_l \wedge y < y_r) \tag{3}$$

where always operator $\Box_I \phi ::= \neg(\top \mathbf{U}_I \neg\phi)$. Quantitative semantics for temporal logics [1] have been proposed for LTL; we include the definition below.

Definition 4 (Robust Satisfaction Value): The robust satisfaction value is a function ρ mapping φ, the trace $\mathbf{x}$, and a sampling period i as follows:

$$\begin{aligned}
\rho(\top, i) &= +\infty \\
\rho(f(\mathbf{x}) > 0, \mathbf{x}, i) &= f(\mathbf{x}_i) \\
\rho(\neg\varphi, \mathbf{x}, i) &= -\rho(\neg\varphi, \mathbf{x}, i) \\
\rho(\varphi_1 \wedge \varphi_2, \mathbf{x}, i) &= \min(\rho(\varphi_1, \mathbf{x}, i), \rho(\varphi_2, \mathbf{x}, i))
\end{aligned}$$

$$\rho\left(\varphi \mathbf{U}_I \psi, \mathbf{x}, i\right) = \sup_{i_1 \in i+I} \min(\rho(\psi, \mathbf{x}, i_1), \inf_{i_2 \in (i, i_1)} \rho(\varphi, \mathbf{x}, i_2)) \quad (4)$$

The robust satisfaction value can be used to measure the distance of the current trace to the error state. The error state could be an unsafe state that may cause a physical safety problem, such as aircraft collision, water overflowing in a water treatment facility [25]. Or the error state is a state that could avoid the attack detection mechanisms [26]. We can convert the stealth attack synthesis problem to a falsification problem if the detection mechanisms could be described by the temporal logic. For example, the false data injection attack could bypass the bad measurement detector by crafting the attack vector to make the difference $||y - \hat{x}||$ between sensor measurement and state estimate below the threshold ϵ. The robust satisfaction value is the difference value $||y - \hat{x}|| - \epsilon$ for this problem. If the robust satisfaction value of the current trace is negative, there is an error state in the trace. To simplify the notions, we reuse the notion $\varphi(\mathbf{x}) = \rho(\varphi, \mathbf{x}, 0)$.

C. ASSUMPTION

Like most of the past work on sensor spoofing attacks [27], [28], we assume that

- the fuzzer knows the configuration of target CPSs, including the sampling period Δt, the time horizon T and the domain of the initial states $\mathcal{X}$ to setup various environments of the physical plant;
- the fuzzer knows the spoofing model's configuration, including the perturbation function *pert* and the domain of the attack signal $\mathcal{Y}'$;
- the full execution trace $\mathbf{x}$ is observable to the fuzzer to monitor the satisfaction of the safety specification.

III. DESIGN

A. OVERVIEW OF CYBER-PHYSICAL FUZZING

The target of falsification is to solve the following optimization problem:

$$\text{Find } (\mathbf{x}_0^*, \Delta\mathbf{y}^*) \text{ such that } \varphi(\mathbf{x}^*) < 0. \quad (5)$$

The design philosophy of CPFuzz is to solve the problem in Equation 5 more efficiently by exploiting more information about the controller program branch coverage. In this section, we illustrate the workflow of CPFuzz shown in Figure 1. CPFuzz is essentially a feedback-guided fuzzing technique. It shares the typical components as the conventional feedback-guided fuzzing techniques. We introduce the following essential techniques to falsify a CPS more efficiently.

- Branch coverage exploring. S-TaLiRo and other similar falsification techniques use robust satisfaction value $\varphi(\mathbf{x}^*)$ as the metric. However, they have not considered the controller program implementation when generating input signals. Our experience shows that the gradient of robust satisfactions value would vanish when falsifying a CPS with a complex controller program. Therefore, we try to include program branch coverage to the metric (Section III-B).
- Structural sensor data representation. Unlike a traditional program, the inputs of CPS, i.e., sensor values, are relatively fixed, such as the location, velocity, temperature, concentration, and images. Because we assume the fuzzer knows the configuration of target CPS, we could exploit this information for initial seeds generation and inputs mutation to effectively explore the vast yet sparse domain of valid program inputs (Section III-C).
- Energy Assignment. To reach the unsafe states more quickly, we need an evaluation function to determine which inputs are likely to be tried next. If one input explores new program branches, it is more likely to generate children with new branch coverage. If one input finds a lower robust satisfaction value, it is more likely to generate children to violate the safety specification. We give these two kinds of inputs, more energy to mutate (Section III-D).
- Mutations. Mutations without the structural information of inputs but may produce invalid inputs and been filtered out by the anomaly detection. Therefore, bit-level mutations, such as bitflip, arithmetic, and other mutation operators in AFL, cannot be used. We define new mutation operators for the sensor inputs and initial states, which are real numbers for most cases (Section III-E).

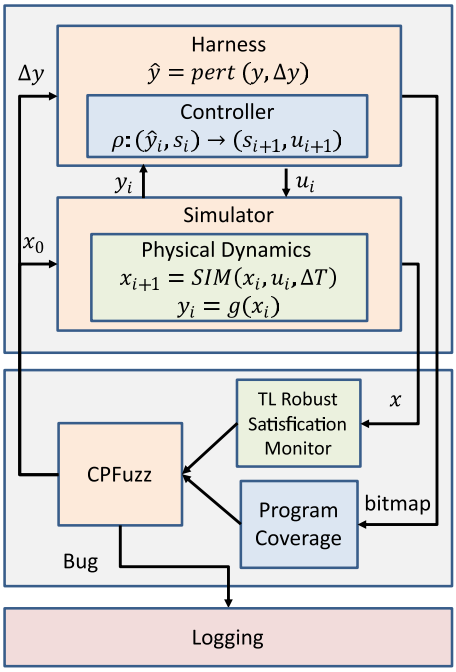


FIGURE 1. Overview of the fuzzing approach. The framework of cyber-physical fuzzing mainly includes three parts: the controller harness, the physical dynamics simulator, and the feedback guided fuzzing loop.

Algorithm 1 Cyber-Physical Fuzzing Loop

Input: The simulation function **SIM**, sensor observer function g, controller function ρ, perturbation funtion *pert*, seed corpus *Seeds*, domain of physical states $\mathcal{X}$, domain of attacking signal $\mathcal{Y}'$ and the safety specification φ.

Output: *Violations*.

1: $Violations \leftarrow \emptyset$
2: $\rho' \leftarrow$ **instrument**(ρ)
3: $S \leftarrow$ (**SIM**, g, ρ', $pert$)
4: **evalSeedAndSaveQueue**(*Seeds*, *Queue*)
5: **repeat**
6: $(\Delta\mathbf{y}, \mathbf{x}_0, bmap, rob_{LTL}) \leftarrow Queue$.**pop**()
7: $score \leftarrow$ **getRobScore**(rob_{LTL})
8: $score \leftarrow$ **getBitmapScore**$(bmap) \cdot score$
9: **for** i from 1 to $score$ **do**
10: $\Delta\mathbf{y}' \leftarrow$ **mutateHavoc**$(\Delta\mathbf{y}, \mathcal{Y}')$
11: $\mathbf{x}_0' \leftarrow$ **mutate**$(\mathbf{x}_0, \mathcal{X})$
12: $(map, robustness) \leftarrow$ **sim**$(\Delta\mathbf{y}', \mathbf{x}_0', S, \varphi)$
13: **if newCoverage**(map) || **newCoverage**$(robustness)$ **then**
14: $Queue$.**add**$(\Delta\mathbf{y}', \mathbf{x}_0', map, robustness)$
15: **end if**
16: **if** $(robustness < 0)$ **then**
17: $Violations \leftarrow Violations \cup input'$
18: **end if**
19: **end for**
20: cross(Queue)
21: **until** Timeout
22: **return** *Violations*

B. ALGORITHM FOR CYBER-PHYSICAL FUZZING

Algorithm 1 shows the high-level functionality of our fuzzing approach.

1) INITIALIZING

First, we instrument the controller program at each branch point and inject a code fragment to record the code locations before the branch and after the branch to help the fuzzer make this judgment on the input's novelty. We group all the programs used for the simulation as S (line 3). Our approach needs to construct a seed corpus of initial state $\mathbf{x}_0$ and input signals $\Delta\mathbf{y} = (\Delta\mathbf{y}_1, \Delta\mathbf{y}_2, \ldots, \Delta\mathbf{y}_N)$, according to the historical data or randomly drawing $\Delta\mathbf{y}_i$ from the predetermined domain of candidate inputs $\mathcal{Y}'$, and the initial states $\mathbf{x}_0$ from the domain $\mathcal{X}$. We also include the boundary values of the domains in the corpus. Initially, the queue of interesting inputs *Queue* is empty, and all seeds are unexplored. We evaluate the seeds by running the inputs and save the results in the queue (line 4).

2) SEED SELECTION

During each iteration of the fuzzing loop, CPFuzz selects the first input in *Queue* (line 6) and searches for new input that explores the solution space until timeout. As the search proceeds, the algorithm adds the input to *Queue* if it finds new coverage or a lower robust satisfaction value. The Queue stores not only the seed input of initial states $\mathbf{x}_0$ and perturbation $\Delta\mathbf{y}$, but also the simulation result *bmap*, rob_{LTL}, which store the branch coverage information and the robust satisfaction value separately.

3) ENERGY ASSIGNMENT

For each seed input $(\mathbf{x}_0, \Delta\mathbf{y})$, the energy determines the number of inputs that are generated by slightly mutating the seed (i.e., energy for $(\mathbf{x}_0, \Delta\mathbf{y})$). We take the function described in Section III-D to map the robust satisfaction value or robustness $rob_{LTL} \in \mathbb{R}$ to the $score \in [1, 100]$ (line 7). Then, we multiply it with the score function implemented in AFL, which evaluate the execution time, branch coverage, and creation time of the seed.

4) MUTATION

Then, the fuzzer generates new inputs by mutating the seed input according to our new mutation operators defined in Section III-E. The `mutateHavoc` function determines where to mutate by choosing an element in the vector, how to mutate by sampling from distribution within the domain. Then we get the new input, $(\mathbf{x}_0', \Delta\mathbf{y}')$ drawing from $\mathcal{X}$ and $\mathcal{Y}'$ by mutating the seed $(\mathbf{x}_0, \Delta\mathbf{y})$ (line 10-11).

5) EXECUTION

Given the input, we can simulate the running of CPS S in a Software-in-the-Loop way and monitor trace in the control loop with the satisfaction value *robustness* of the safety specification φ (line 12). The initial state $\mathbf{x}_0$ is sent to the simulator to set up the initial configuration, such as the positions and velocities of objects. The adversarial perturbation $\Delta\mathbf{y}$ is sent to the controller harness, which generates the final input to the controller and gets the coverage information *map* while running the instrumented version of the controller.

6) LOGGING AND CROSSOVER

The *Queue* is updated if we find an interesting input, that has a new branch coverage and lower robustness (line 13-15). If $robustness < 0$, we find a violation of the safety specification φ and record it in the set *Violations* with the result information. After exhausting the seed's energy, we crossover the current seed with a random seed in the *Queue* to save useful features of the seed (line 20). In AFL, this is implemented by splicing two distinct inputs at a random location, but it will damage the seed structure in our scenario. Therefore, we use the simulated binary crossover (SBX) [29] on a randomly chosen valuable to combine two seeds. SBX is usually used in evolutionary algorithms for local searching.

C. INPUT STRUCTURE AND ATTACK SIMULATION

We separate input of CPS simulation into two parts: the initial state $\mathbf{x}_0$ and the adversarial perturbation $\Delta\mathbf{y}_i, i \in 0..N-1$.

Algorithm 2 Cyber-Physical System Simulation (**sim**)

Input: The simulation function **SIM**, sensor observer function g, instrumented version of controller function ρ', perturbation funtion *pert*, initial physical states $\mathbf{x}_0$, attacking signal $\Delta\mathbf{y}$ and the safety specification φ.

Output: The branch coverage bitmap *map* and LTL robust satisfaction value *robustness*.

1: $map \leftarrow \emptyset$
2: **for** i from 1 to N **do**
3: $\quad \hat{\mathbf{y}}_i \leftarrow pert(g(\mathbf{x}_i), \Delta\mathbf{y}_i)$
4: $\quad (\mathbf{s}_{i+1}, \mathbf{u}_i, map') \leftarrow \rho'(\hat{\mathbf{y}}_i, \mathbf{s}_i)$
5: $\quad map \leftarrow map \cup map'$
6: $\quad \mathbf{x}_{i+1} \leftarrow \mathbf{SIM}(\mathbf{x}_i, \mathbf{u}_i, i)$
7: **end for**
8: $robustness \leftarrow \varphi(\mathbf{x})$
9: **return** $(map, robustness)$

The initial state is a vector representing the initial physical states, like the positions, velocities, headings of the vehicles in a scenario. The range of each component $\mathcal{X}_i$ is predefined. The adversarial perturbation is a vector representing the attack parameters in a sampling period. In each iteration of the Algorithm 2, the harness read the adversarial perturbation $\Delta\mathbf{y}_i$ from the fuzzer and the sensor value $\mathbf{y}_i$ from the controller to generate the finial sensor value $\hat{\mathbf{y}}_i$ for this iteration (line 3). The function *pert* is given by the users according to the threat model. Different sensor models have different perturbation functions *pert*. Here are some examples of the perturbation functions.

- In smart grid, $pert(\mathbf{y}_i, \Delta\mathbf{y}_i) = \mathbf{y}_i + \Delta\mathbf{y}_i$, where $\mathbf{y}_i$ is meter measurements of power and the $\Delta\mathbf{y}_i$ is the injected false data directly applied on the measurements [30]. Most kinds of spoofing attacks can adopt this perturbation function.
- GPS measures the signal transmitting time $\mathbf{y}_i$ between the satellites and the target to estimate the position. GPS spoofing can only delay the authenticated signal [31], therefore $pert(\mathbf{y}_i, \Delta\mathbf{y}_i) = max(\mathbf{y}_i, \Delta\mathbf{y}_i)$.
- LiDAR can also be a target for sensor spoofing. Attacker can inject spoofed 3D point cloud $\Delta\mathbf{y}_i$ into the original 3D point cloud $\mathbf{y}_i$, and merge it $pert(\mathbf{y}_i, \Delta\mathbf{y}_i) = \mathbf{y}_i \oplus \Delta\mathbf{y}_i$, where $\oplus$ is merge function defined in [32].

The dimensions of the sensor input and the perturbation can be different, defined in the threat model's perturbation function. In more realistic attack scenarios, the attacker has limited capabilities to effect all the sensors.

The *map* record how many times one branch is taken. Initially, the *map* is empty (line 1). In each iteration, the harness executes the controller program ρ' given the input $\hat{\mathbf{y}}_i$ and controller state $\mathbf{s}_i$. Then, we can get the next controller state $\mathbf{s}_{i+1}$, controller's output $\mathbf{u}_i$ and the new coverage information map'. The controller state $\mathbf{s}_i$ is used to hold the historical information about the controller between two iterations. After updating the coverage information, the harness would query the simulator for the new state $\mathbf{x}_{i+1}$ by sending current state $\mathbf{x}_i$, controller output $\mathbf{u}$ and the iteration time i. In the end, the harness could get the *robustness* by applying robust satisfaction semantics of φ on the trace $\mathbf{x}$ and the coverage information *map* of the whole execution.

D. ENERGY ASSIGNMENT

Energy assigned to seed is the number of offspring generated from the seed. The more interesting the seed is, the higher the energy should be. In cyber-physical fuzzing, energy is a multiplication of the AFL function calculate_score evaluating the program execution performance (prioritize inputs that cover more paths) and the robustness score evaluating the safety specification (prioritize inputs that make system unsafe). The robustness is a real number that measures the satisfaction of the safety specification, ranging from $-\infty$ to ∞. However, the AFL score ranges from 2 to 1600 by default, so we cannot multiply the robustness directly. To make the final energy reasonable, we define multiplier s_r, the score for robustness as the following:

$$s_r = \begin{cases} \lceil S_{max} \cdot e^{-r} \rceil & r < R_{min} \\ \lceil \dfrac{S_{max} \cdot (R_{max} - r)}{R_{max} - R_{min}} \cdot e^{-r} \rceil & R_{min} \le r < R_{max} \\ 1 & r \ge R_{max} \end{cases} \tag{6}$$

where r is the robustness of the safety specification, R_{min} and R_{max} denote the minimum and maximum value of the robustness for all evaluated inputs. After multiplying the AFL score and the robustness score s_r, we compare it with the maximum score 1600 in AFL and keep the smaller one.

The robustness score s_r is computed depending on two aspects, the current robustness, and the historical statistics. If $rob > R_{max}$, we do not want to amplify the original AFL score. The seed is more likely added because it found a new branch statement. If $R_{min} \le rob < R_{max}$, the score depends on how close it is to the lowest robustness and the robustness itself. e^{-rob} makes the score decade as *rob* increases. If $rob < R_{min}$, a new lowest robustness is founded, and violations are more likely generated from this seed input. It is possible that $s_r > S_{max}$ when $rob < 0$, so AFL limits the energy within the maximum value.

E. MUTATIONS

The effectiveness of cyber-physical fuzzing also comes from the careful design of its mutation operators. These operators should fully leverage the input domain information in the user-specific configuration to generate new inputs within the domain. They should also make sure that after a limited number of mutation operators, any input within the domain can be reachable.

As shown in Algorithm 3, the number of mutation operators is randomly chosen between 1 and the length of $\mathbf{v}$. For each mutation operator, one element of the input is changed. More mutation operators are applied, more different there are between the seed $\mathbf{v}$ and $\mathbf{v}'$. To mutate one element,

Algorithm 3 Multi-Dimensional Mutation for Real Numbers (**mutateHavoc**)

Input: The input vector $\mathbf{v}$ and its domain $\mathcal{D}$.
Output: mutated vector $\mathbf{v}'$.

1: $numMutation \leftarrow$ **randomBetween**$(1, |\mathbf{v}|)$
2: $\mathbf{v}' \leftarrow \mathbf{v}$
3: **for** i from 1 to $numMutation$ **do**
4: $\quad k \leftarrow$ **randomBetween**$(1, |\mathbf{v}|)$
5: $\quad \mathbf{v}' \leftarrow$ **mutate**$(\mathbf{v}', k, \mathcal{D})$
6: **end for**
7: $\mathbf{v}' \leftarrow$ **clip**$(\mathbf{v}', \mathcal{D})$
8: **return** $\mathbf{v}'$

and the element index in the input vector k is randomly chosen, which determines how to mutate and which input value to mutate.

There are three types of mutation operators in the cyber-physical fuzzing, which are borrowed from real space mutations in evolutionary computing [33]. The first one is uniform mutation operator, which draws a real number v'_k in the interval $\mathcal{D}_k = [l_k, u_k]$ of the k-th component of the input. The second one is the Gaussian mutation operator, which draws a real number in Gaussian distribution $v'_k = N(v_k, \sigma)$. σ stands for the mutation step. The third one is non-uniform mutation operator, which is defined as follow:

$$v'_k = \begin{cases} v_k + \Delta(t, u_k - v_k) & \gamma = 0 \\ v_k - \Delta(t, v_k - l_k) & \gamma = 1 \end{cases} \tag{7}$$

where γ chooses 0 or 1 randomly, t is the iteration of the evolutionary fuzzing loop, $\Delta(t, y)$ return a value in $[0, y]$, such that as t increases, the $\Delta(t, y)$ is more possible to approaching 0. The non-uniform mutation operator narrows the range of the mutation as the evolution proceeds.

The v' after the second or third mutation operators may out of the domain $\mathcal{D}$, and we clip the result to maintaining validity (line 7).

IV. IMPLEMENTATION

We evaluate the effectiveness of Cyber-Physical fuzzing by implementing a prototype tool, CPFuzz. As shown in Figure 1, the overall architecture of CPFuzz contains two parts: the simulation part and the main fuzzer.

The simulation part also includes two components: the harness and the simulator. The harness implemented in C loads the controller program and starts the control loop according to the configurations. The simulator implemented in Python loads the physical dynamics engine and waits for the controller to sample the measurements $\mathbf{y}_i$ of the plant. They communicate the sensor data and controller output through *socket*.

The fuzzer part of CPFuzz extends AFL by adding and modifying four components, the Configuration Parser, the Mutation Operators, the LTL Robustness Satisfaction Calculator, and the Energy Assignment. The fuzzer get the trace and bitmap information from the harness through the shared memory. Our CPFuzz prototype implementation and benchmarks are open source at GitHub.[1]

TABLE 1. The information about the benchmarks.

Benchmark	ΔT	Controller			Plant	
		Pert.I	LOC	Paths	Modes	States
Heater	0.2	0	59	26	3	1
Heat	0.5	0	37	312	6	3
DC Motor	0.02	1	29	3	0	2
FuzzyC	0.01	0	218	208	0	3
SPI	1	1	13	3	1	1

V. EXPERIMENTAL EVALUATION

The experiments have two goals. Firstly, in Section V-B, we evaluate the impact of different choices of parameters for CPFuzz (such as the mutation operators and robustness score parameters). Secondly, in Section V-C, we evaluate the performance of CPFuzz in comparison to the state-of-the-art.

A. BENCHMARKS

We consider five case studies: a heater system that uses a thermostat to switch operating modes to maintain a comfortable temperature in a room, a heat benchmark shuffle a limited number of heaters to maintain a comfortable temperature in all rooms, a closed-loop model of the DC motor with controller disturbance of armature current and angular velocity, a nonlinear inverted pendulum balanced on a cart using rule-based controllers and an academic model of sampled polarity integrator system that measures how much longer a signal remains positive than it remains negative. The related information about the benchmarks is concluded in Table 1. For each model, we try to falsify given LTL safety specifications listed in Table 2.

1) HEATER [5]

The heater system consists of a room, and a heater controlled by a thermostat. The heater has 3 operating modes; off, regular heating, and fast heating. The controller has built-in logic to prevent chattering, i.e., avoiding rapid switching of the heater between modes. The heater is modeled as a hybrid system with linear dynamics, with one continuous state and three modes. We try to falsify the property that the room-temperature is always greater than 52° F.

2) HEAT BENCHMARKS [12]

The heat benchmarks have a limited number of heaters h used to heat rooms r, where $h < r$. We choose the first instance, which has 3 rooms and 1 heater. Correspondingly, the plant has 3 continuous states and 6 modes. The heater's location characterizes each mode, and it is a discrete state (on/off). We try to falsify the property that the first room's temperature does not drop below 17.23 °C.

[1] https://github.com/shangfute/CPFuzz

TABLE 2. The comparison of three mutation operators. All processes were run as single threaded on Ubuntu 14.04, running on an Intel i7-9750H CPU @2.60GHz with 8GB RAM. All times are in seconds(s).

Benchmark	Specification T(s)	Uniform	Non-uniform	Gaussian
Heater	$\square_{[0,2]}(x > 52)$	178	40.2	**30**
Heat	$\square_{[0,10]}(x_0 > 17.23)$	87	**20**	47
DC Motor	$\square_{[0,1]}\neg(x_0 \in [1, 1.2] \wedge x_1 \in [10, 11])$	8	**6**	8
FuzzyC	$\square_{[0,0.1]}\neg(x_0 \in [-4, 4] \wedge x_1 \in [1.5, 10] \wedge x_2 \in [-20, 20])$	151	34	**4**
SPI(P1)	$\square_{[0,50]}(x < 20)$	41	15	**7**

3) DC MOTOR [5]

The DC motor is a linear continuous system with armature current and angular velocity. The bounded additive perturbation Δy in the controller induces error in the sensed plant outputs. It's dangerous that armature current and angular velocity enter the unsafe state. We try to falsify the property that the system never enter the region of the state-space: $x_0 \in [1, 1.2], x_1 \in [10, 11]$.

4) FUZZY CONTROL OF INVERTED PENDULUM [13]

The fuzzy controller tries to stabilize a nonlinear inverted pendulum balanced on a cart. The controller classifies the current plant state and selecting a corresponding control output from a lookup table. If the position, velocity and acceleration of the pendulum enter a certain area, the control system fails. We try to falsify the property that the system never enter the following region of the state-space $x_0 \in [-4, 4] \wedge x_1 \in [1.5, 10] \wedge x_2 \in [-20, 20]$.

5) SAMPLED POLARITY INTEGRATOR SYSTEM [11]

The sampled polarity integrator system has a perturbation input $\Delta y \in [-1, 1]$, and a single continuous state x. The controller outputs $u = sign(\Delta y)$. The continuous state of the plant evolves as $\dot{x} = u$. We try to falsify the property that P1: $x < 20$, P2: $x < 50$ and P3: $x < 150$ for time horizons 50, 200, and 500 respectively. This system is a academic model to evaluate the efficiency of fuzzing process. The difficulty of identifying violations can be easily adjusted by modifying the specification.

B. EVALUATION OF MUTATION OPERATORS CHOICES

We compare the performance of different mutation operators for each test case. The fuzzers with different mutation operators are running on an Intel i7-9750H CPU @2.60GHz with 8GB RAM and keep the other parts of CPFuzz the same. We set the mutation step $\sigma = 1$ in the Gaussian Mutation operator. And we set $\Delta(t, y) = y \times (1 - r^{(1-t/T)^2})$, $T = 1000$, r is randomly chosen in [0, 1] in the non-uniform mutation operator.

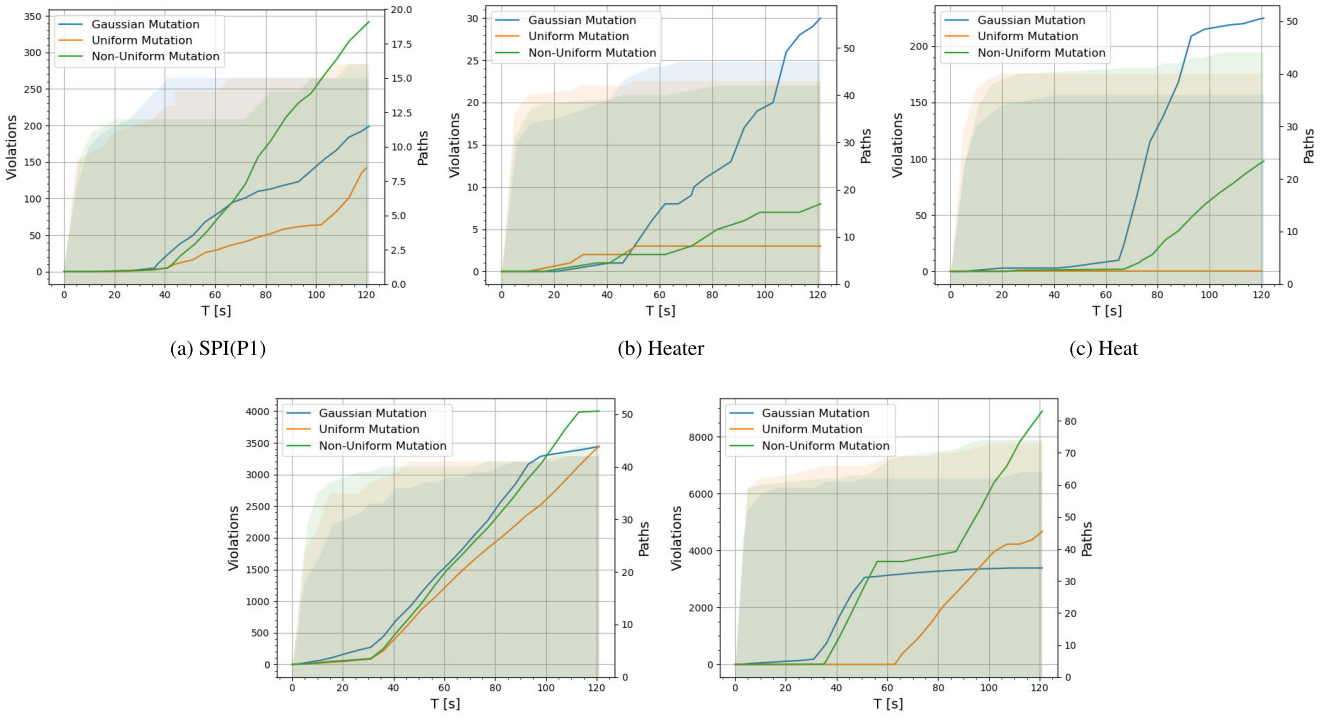
To determine which mutation operator is most effective, we evaluate the mutations from two aspects: how fast to find a violation, and how many violations to find in two minutes. The results in the first aspect of the study are summarized in Table 2. Note that for each item in the table, an average execution time over 10 runs is reported. Average execution time is required due to the randomized nature of all three. We can find that the uniform mutation operator behaves the worst in all benchmarks. The uniform mutation operator can efficiently generate valid inputs, but it does not utilize the parent's information and search the entire space uniformly. The Gaussian mutation operator finds violations more quickly than the non-uniform mutation operator in the most benchmarks. The Gaussian mutation operator has better performance in **local** searching according to the principle of maximum entropy.

Fig. 2 shows the results in the second aspect of the study, which describe the violations found by CPFuzz in two minutes. The uniform mutation operator finds the least violations in most benchmarks. Gaussian mutation operators still found more violations in benchmarks Heater and Heat. However, in benchmark SPI(P1), DC Motor, and FuzzyC, the non-uniform mutation operator found more violations. Even though in these three benchmarks, the Gaussian mutation operator found the first 10 violations faster than the non-uniform mutation operator, which means that the non-uniform mutation operator has better performance in **global** searching. It is better to combine the advantage of the non-uniform mutation operator and the Gaussian mutation operator to have a balanced performance in mutation. We randomly choose one mutation operator from these two, and the results are demonstrated in Table 3.

From the coverage information shown in the colored areas in Fig. 2, most program paths are explored before CPFuzz finding the first violations. There is a positive relationship between the explored paths and found violations over time. The more program paths CPFuzz explored, the more possibility to find violations. This result proves we can solve the falsification problem by exploiting the fuzzing, a promising technique to find program bugs.

C. EVALUATION OF PERFORMANCE

Our approach combines the idea of the fuzzing and falsification, so we hope to compare CPFuzz with state-of-art tools in fuzzing and falsification. In falsification tools, we choose the mature tools, S-TaLiRo [14], the falsifier works with industry-standard models and the S3CAMX [5], the falsifier uses static symbolic execution on the controller program to find violations. However, most fuzzers are not supported to find safety violations in CPS. Therefore, we set the robustness score $s_r = 1$ and keep other parts of the CPFuzz, such as the simulation engine and the mutation operators. Then, the fuzzer will not utilize the safety specification to guide our



(a) SPI(P1) (b) Heater (c) Heat

(d) DC Motor (e) FuzzyC

FIGURE 2. The relation between paths and violations of different benchmarks in a limited time (two minutes). The colored areas describe the explored program paths over time. The lines represent the violations found over time. The performance of the Gaussian mutation operator is colored blue, the non-uniform mutation operator is colored green, and the uniform mutation operator is colored orange. Most paths are covered before finding the first violations, and the more paths are covered, the more violations are founded.

TABLE 3. The comparison of CPFuzz, S3CAMX, S-Taliro, and the real number version of AFL. All processes were run as single threaded on Ubuntu 14.04, running on an Intel i7-9750H CPU @2.60GHz with 8GB RAM. All times are in minutes.

Benchmark	Specification T(s)	AFL	S-TaLiRo	S3CAMX	CPFuzz
Heater	$\square_{[0,2]}(x > 52)$	1.36	0.45	0.18	**0.13**
Heat	$\square_{[0,10]}(x_0 > 17.23)$	3.37	4.32	0.18	**0.11**
DC Motor	$\square_{[0,1]}\neg(x_0 \in [1, 1.2] \wedge x_1 \in [10, 11])$	0.16	**TO**	0.91	**0.08**
FuzzyC	$\square_{[0,0.1]}\neg(x_0 \in [-4, 4] \wedge x_1 \in [1.5, 10] \wedge x_2 \in [-20, 20])$	4.52	0.22	1.33	**0.17**
SPI(P1)	$\square_{[0,50]}(x < 20)$	1.12	0.29	0.19	**0.06**
SPI(P2)	$\square_{[0,200]}(x < 50)$	7.26	16.26	1.08	**0.28**
SPI(P3)	$\square_{[0,500]}(x < 150)$	**TO**	**TO**	**6.35**	**TO**

search, just focus on improving the program coverage. For each benchmark, AFL, S-TaLiRo, S3CAMX, and CPFuzz run for 10 times on the same configuration. If all 10 run takes more than 1 hour to finish, we consider it a time out (TO).

As shown in Table 3, CPFuzz has a better performance than AFL and S-TaLiRo, which generate input by random sampling but have different optimization metrics. On average of 10 runs, CPFuzz reaches an unsafe state 10X, 31X, 2X, 27X, 18X, and 26X faster than AFL on Heater, Heat, DC Motor, Fuzzy Controller, SPI(P1), and SPI(P2) respectively. CPFuzz is about 3X, 39X, ∞X, 1.2X, 5X, and 58X faster than S-TaLiRo on Heater, Heat, DC Motor, and Fuzzy Controller, SPI(P1), and SPI(P2) respectively. This result is encouraging as it shows the ability of CPFuzz to analyze a large number of control-flow paths and be successful at finding a falsifying trajectory.

Compared with constraint-solver based S3CAMX, for benchmark Heat, Heater, DC Motor and Fuzzy Controller, SPI(P1), and SPI(P2), the CPFuzz could find violations in less time. In SPI(P3), the S3CAMX has a better performance. Next, we analyze the reason why CPFuzz fails on benchmark SPI(P3). In the SPI benchmarks, we observe that in order to falsify the requirement $\square_{[0,N]}(x < k)$, the required input Δy would have to be positive at over $\lceil\frac{N+k}{2}\rceil$ control iterations. The probability of at least $\lceil\frac{N+k}{2}\rceil$ of uniform-randomly chosen numbers in the interval $[-1, 1]$ being positive is tiny (when k is comparable to N). In P2 $k = 50, N = 200$, the possibility to reach an unsafe state is $C(200, 125)(1/2)^{200} \approx 0.0001$. When it comes to P3 $k = 150, N = 500$, the possibility to reach an unsafe state is $C(500, 325)(1/2)^{500} \approx 4 \times 10^{-12}$. CPFuzz has to utilize the coverage and the robustness to guide the

search. However, the SPI benchmark has only three paths, which means CPFuzz can hardly optimize the search by the coverage information. Simultaneously, the score calculated based on the program branch information may affect the energy assignment to search for some non-targeted inputs and slow down the search speed. However, this problem could be solved by solving the constraints by static symbolic execution in S3CAMX. There is considerable research on hybrid fuzzing [34], [35] about applying dynamic symbolic execution in the fuzzing. It could be a promising research direction to apply hybrid fuzzing in cyber-physical fuzzing.

D. IDENTIFY IMPLEMENTATION VULNERABILITIES

We have discussed the CPFuzz could be used to find design flaws like traditional falsification tools. Consider the example of a fixed-point overflow vulnerability in controller implementation, which could not be detected by classical fuzzing or falsification. In this case study, we implement the DC motor example under a fixed-point representation of real numbers. Fixed-point arithmetics is computationally less expensive than floating-point arithmetics and therefore is still applied in many nowadays embedded systems. The DC motor system has two state variables: armature current x_0 and angular velocity x_1. The sensor is designed to sense the x_0, x_1 after every $\Delta = 0.02$s. The plant dynamics are modeled as a single-mode hybrid automaton with linear dynamics, as shown in Fig. 3. The control software is a C program shown in Fig. 4. The controller input $y = x_0$ is the sensor value, and attack $= \Delta y \in [-0.5, 0]$ is the perturbation added on the sensor value. error_i_prev is the accumulated error used for the PI controller. The PI controller's target is to control the armature current x_0 to the reference value of 1. Our target is to find a sequence of sensor disturbances that lead to a violation. The system starts with $x_0 = 0, x_1 = 0$ should never enter the region $x_0 \in [1.2, 1.4], x_1 \in [13, 15]$ within 1s of system operation.

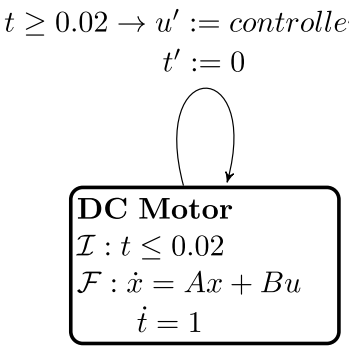


FIGURE 3. The hybrid automaton for the DC motor, where $A = \begin{bmatrix} -10 & 1 \\ -0.02 & -2 \end{bmatrix}$, $B = \begin{bmatrix} 0 & -100 \\ 2 & 0 \end{bmatrix}$.

We used 64-bit signed fixed-point representation for all the variables in the program. We adjusted the integer precision using a counterexample-guided loop to find violations very hard in a long timeout. We used 40 bits for the integer part and the remaining 23 bits for the fractional part in the case study. The addition and multiplication computation are implemented under fixed-point arithmetics. The controller output pid_op $= u$ is limited between [SAT, $-$SAT] which results in 3 control-flow paths.

Our implementation runs for about 2 seconds to discover a violation as shown in Fig. 5. We validated the attack

```
#define SAT (20.0)
#define UPPER_SAT (SAT)
#define LOWER_SAT (-SAT)

fixedpoint KP = 40.0;
fixedpoint KI = 1.0;
fixedpoint ref = 1.0;
fixedpoint error_i_prev = 0.0;

fixedpoint controller(fixedpoint y,
                      fixedpoint attack)
{
  fixedpoint spoofed, error, error_i;
  fixedpoint pid_op = 0.0;

  // spoofing attack sensor
  spoofed = y + attack;
  error = ref - spoofed;
  error_i = error * KI + error_i_prev;
  error_i_prev = error_i;
  pid_op = error * KP + error_i * KI;

  if(pid_op > UPPER_SAT)
    pid_op = UPPER_SAT;
  else if(pid_op < LOWER_SAT)
    pid_op = LOWER_SAT;
  else
    pid_op = pid_op;
  return pid_op;
}
```

FIGURE 4. Program implementation of the DC motor PI controller.

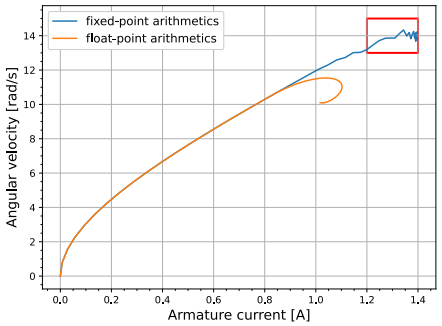


FIGURE 5. Validation of the same attack input on the DC motor system with two real number representations in 1s. Red rectangle denotes the boundary of the unsafe region. The trace under fixed-point arithmetics enters unsafe region due to fixed-point overflow.

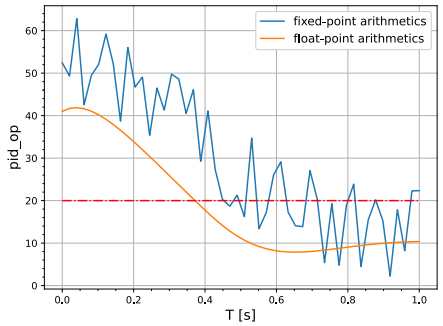


FIGURE 6. An illustration of the pid_op under the same attack vector with two real number representations. The chatter of the trace under fixed-point arithmetics is caused by fixed-point overflow.

sequences under the double-precision float-point representation. We can find that the controller under float-point arithmetics could control the armature current back to the reference value. The trace under fixed-point arithmetics entered the unsafe region under the same attack vector. To discover the reason for differences between two results, we visualize the traces of controller output within 1s of system operation. Fig. 6 demonstrates the controller output pid_op before limited by the maximum value $SAT = 20$. The pid_op under float-point arithmetics is relatively smooth over time. However, pid_op under fixed-point arithmetics has obvious chattering. Before 0.35s, two traces are both above 20, and the controller output is the same, which results in the overlay part of the state trace. The average value of pid_op under fixed-point arithmetics in a short window is greater than the value under float-point arithmetics, which causes the armature current to continue increasing over the reference value. The root cause of violation is the fixed-point overflow during computation, and non-saturated signed fixed-point arithmetics, which could not be identified by AFL or libfuzzer. The falsification tools, like S-TaLiRo and S3CAMX, also haven't considered the fixed-point arithmetics. Boolean satisfiability (SAT) solver could be used to synthesis an attack sequence by encoding the program with fixed-point semantics into a Boolean satisfiability problem [7]. However, our approach is more flexible and could be applied to other vulnerabilities under a grey-box program model.

VI. CONCLUSION

We introduced the concept of coverage-guided fuzzing for CPS falsification and demonstrated the practical applicability of CPFuzz, a novel fuzzer in cyber-physical system testing. The key idea is to model not only the controller program but also the physical environment and combine the characteristics of evolutionary fuzzing with branch coverage and LTL robustness to balance the exploration-exploitation trade-off in cyber and physical space. We ran CPFuzz against a set of control system benchmarks and compared it with state-of-the-art fuzzing, falsification tools for CPS. The results demonstrated the effectiveness of applying fuzzing in the falsification problem of CPS. Because of the similarities between fuzzing and falsification, many ideas in fuzzing could be applied to the falsification problem. In future work, we plan to improve code coverage by dealing with more sophisticated constraints by exploiting the gradient information to guide the mutation direction.

REFERENCES

[1] J. V. Deshmukh, A. Donzé, S. Ghosh, X. Jin, G. Juniwal, and S. A. Seshia, "Robust online monitoring of signal temporal logic," 2015, *arXiv:1506.08234*. [Online]. Available: http://arxiv.org/abs/1506.08234

[2] Y. S. R. Annapureddy and G. E. Fainekos, "Ant colonies for temporal logic falsification of hybrid systems," in *Proc. 36th Annu. Conf. IEEE Ind. Electron. Soc. IECON*, Nov. 2010, pp. 91–96.

[3] H. Abbas and G. Fainekos, "Convergence proofs for simulated annealing falsification of safety properties," in *Proc. 50th Annu. Allerton Conf. Commun., Control, Comput. (Allerton)*, Oct. 2012, pp. 1594–1601.

[4] Z. Zhang, I. Hasuo, and P. Arcaini, "Multi-armed bandits for Boolean connectives in hybrid system falsification (Extended Version)," 2019, *arXiv:1905.07549*. [Online]. Available: http://arxiv.org/abs/1905.07549

[5] A. Zutshi, S. Sankaranarayanan, J. V. Deshmukh, and X. Jin, "Symbolic-numeric reachability analysis of closed-loop control software," in *Proc. 19th Int. Conf. Hybrid Syst., Comput. Control HSCC*, 2016, pp. 135–144.

[6] J. Kapinski, J. V. Deshmukh, X. Jin, H. Ito, and K. Butts, "Simulation-based approaches for verification of embedded control systems: An overview of traditional and advanced modeling, testing, and verification techniques," *IEEE Control Syst.*, vol. 36, no. 6, pp. 45–64, Dec. 2016.

[7] O. Inverso, A. Bemporad, and M. Tribastone, "SAT-based synthesis of spoofing attacks in cyber-physical control systems," in *Proc. ACM/IEEE 9th Int. Conf. Cyber-Phys. Syst. (ICCPS)*, Apr. 2018, pp. 1–9.

[8] C. S. Pasareanu, J. Schumann, P. Mehlitz, M. Lowry, G. Karsai, H. Nine, and S. Neema, "Model based analysis and test generation for flight software," in *Proc. 3rd IEEE Int. Conf. Space Mission Challenges Inf. Technol.*, Jul. 2009, pp. 83–90.

[9] R. Majumdar, I. Saha, K. C. Shashidhar, and Z. Wang, "CLSE: Closed-loop symbolic execution," in *NASA Formal Methods*, vol. 7226. Berlin, Germany: Springer, 2012, pp. 356–370.

[10] *American Fuzzy Lop*. Accessed: Oct. 1, 2019. [Online]. Available: https://lcamtuf.coredump.cx/afl/, doi: 10.1109/ACCESS.2019.2903291.

[11] T. Dreossi, T. Dang, A. Donzé, J. Kapinski, X. Jin, and J. V. Deshmukh, "Efficient guiding strategies for testing of temporal properties of hybrid systems," in *NASA Formal Methods* (Lecture Notes in Computer Science). Cham, Switzerland: Springer, 2015, pp. 127–142.

[12] A. Fehnker and F. Ivančić, "Benchmarks for Hybrid Systems Verification," in *Hybrid Systems, Computation and Control*, vol. 2993. Berlin, Germany: Springer, 2004, pp. 326–341.

[13] K. M. Passino and S. Yurkovich, *Fuzzy Control*. Menlo Park, CA, USA: Addison-Wesley, 1998.

[14] Y. Annpureddy, C. Liu, G. Fainekos, and S. Sankaranarayanan, "S-TaLiRo: A tool for temporal logic falsification for hybrid systems," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 6605. Berlin, Germany: Springer, 2001, pp. 254–257.

[15] A. A. Julius, G. E. Fainekos, M. Anand, I. Lee, and G. J. Pappas, "Robust test generation and coverage for hybrid systems," in *Hybrid Systems, Computation and Control*, vol. 4416. Berlin, Germany: Springer, 2007, pp. 329–342.

[16] T. Nahhal and T. Dang, "Test coverage for continuous and hybrid systems," in *Computer Aided Verification*, vol. 4590. Berlin, Germany: Springer, 2007, pp. 449–462.

[17] A. Adimoolam, T. Dang, A. Donzé, J. Kapinski, and X. Jin, "Classification and coverage-based falsification for embedded control systems," in *Computer Aided Verification—CAV* (Lecture Notes in Computer Science), vol. 10426, R. Majumdar and V. Kunčak, Eds. Cham, Switzerland: Springer, 2017.

[18] T. C. H. Kim Kim, J. Rhee, F. Fei, Z. Tu, G. Walkuop, and D. Xu, "RVFUZZER: Finding input validation bugs in robotic vehicles through control-guided testing," in *Proc. 28th USENIX Secur. Symp. (USENIX Secur.)*, San Francisco, CA, USA, Aug. 2019, pp. 425–442.

[19] Y. Chen, C. M. Poskitt, J. Sun, S. Adepu, and F. Zhang, "Learning-guided network fuzzing for testing cyber-physical system defences," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 962–973.
[20] Y. Chen, B. Xuan, C. M. Poskitt, J. Sun, and F. Zhang, "Active fuzzing for testing and securing cyber-physical systems," 2020, *arXiv:2005.14124*. [Online]. Available: http://arxiv.org/abs/2005.14124
[21] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 1032–1043.
[22] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 1–16.
[23] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: Efficient fuzzing with neural program smoothing," 2018, *arXiv:1807.05620*. [Online]. Available: http://arxiv.org/abs/1807.05620
[24] E. Plaku, L. E. Kavraki, and M. Y. Vardi, "Falsification of LTL safety properties in hybrid systems," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 5505. Berlin, Germany: Springer, 2009, pp. 368–382.
[25] S. Adepu, N. K. Kandasamy, and A. Mathur, "EPIC: An electric power testbed for research and training in cyber physical systems security," in *Computer Security—SECPRE* (Lecture Notes in Computer Science), vol. 11387, S. Katsikas *et al.*, Eds. Cham, Switzerland: Springer, 2019.
[26] S. Adepu and A. Mathur, "Assessing the effectiveness of attack detection at a hackfest on industrial control systems," *IEEE Trans. Sustain. Comput.*, early access, Oct. 30, 2018, doi: 10.1109/TSUSC.2018.2878597.
[27] S. Amin, X. Litrico, S. S. Sastry, and A. M. Bayen, "Stealthy deception attacks on water SCADA systems," in *Proc. 13th ACM Int. Conf. Hybrid Syst., Comput. Control HSCC*, 2010, pp. 161–170.
[28] J. Goppert, A. Shull, N. Sathyamoorthy, W. Liu, I. Hwang, and H. Aldridge, "Software/hardware-in-the-loop analysis of cyberattacks on unmanned aerial systems," *J. Aerosp. Inf. Syst.*, vol. 11, no. 5, pp. 337–343, May 2014.
[29] K. Deb, K. Sindhya, and T. Okabe, "Self-adaptive simulated binary crossover for real-parameter optimization," in *Proc. 9th Annu. Conf. Genetic Evol. Comput. GECCO*, 2007, p. 1187.
[30] Y. Liu, P. Ning, and M. K. Reiter, "False data injection attacks against state estimation in electric power grids," *ACM Trans. Inf. Syst. Secur.*, vol. 14, no. 13, pp. 1–33, May 2011.
[31] N. O. Tippenhauer, C. Pöpper, K. B. Rasmussen, and S. Capkun, "On the requirements for successful GPS spoofing attacks," in *Proc. 18th ACM Conf. Comput. Commun. Secur. (CCS)*. New York, NY, USA: Association for Computing Machinery, 2011, pp. 75–86.
[32] Y. Cao, C. Xiao, B. Cyr, Y. Zhou, W. Park, S. Rampazzi, Q. A. Chen, K. Fu, and Z. M. Mao, "Adversarial sensor attack on LiDAR-based perception in autonomous driving," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, p. 15.
[33] D. M. Rocke and Z. Michalewicz, "Genetic algorithms + data structures = evolution programs," *J. Amer. Stat. Assoc.*, vol. 95, p. 347, Mar. 2000.
[34] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *Proc. 27th USENIX Secur. Symp. (USENIX Secur.)*, Baltimore, MD, USA: USENIX Association, 2018, pp. 745–761.
[35] 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 *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–16.

FUTE SHANG received the M.S. degree in information and communication engineering from Air Force Engineering University, in 2016, where he is currently pursuing the Ph.D. degree. His current research interest includes cyber-physical system security.

BUHONG WANG received the M.S. and Ph.D. degrees in signal and information processing from Xidian University, Xi'an, China, in 2000 and 2003, respectively. His Ph.D. degree thesis On Some Crucial Aspects of High-Resolution Direction of Arrival Estimation was honored as the Excellent Doctoral Dissertation of Shaanxi Province by Xidian University. From 2003 to 2005, with the support of the National Postdoctoral Science Foundation, he was a Postdoctoral Fellow with the Postdoctoral Technical Innovation Center, Nanjing Research Institute of Electronics Technology, Nanjing, China. From 2006 to 2008, he was an Associate Professor with the School of Electronic Engineering, Xidian University. From 2009 to 2010, he was a Research Fellow with the Department of Electrical and Computer Engineering, National University of Singapore, Singapore. Since 2012, he has been a Professor with the Information and Navigation College, Air Force Engineering University, Xi'an. His current research interests include cyber security and cyber physical systems.

TENGYAO LI received the M.S. degree in computer science and technology from Air Force Engineering University, in 2016, where he is currently pursuing the Ph.D. degree. His current research interests include attack detection and resilient recovery on ADS-B data.

JIWEI TIAN received the M.S. degree in information and communication engineering from Air Force Engineering University, in 2017, where he is currently pursuing the Ph.D. degree. His current research interest includes adversarial attack on cyber physical systems.

KUNRUI CAO is currently pursuing the Ph.D. degree with Air Force Engineering University. He is also a Lecturer with the School of Information and Communications, National University of Defense Technology. His current research interests include the IoT security and physical layer security of wireless communication.