Cyber-Physical Inconsistency Vulnerability Identification for Safety Checks in Robotic Vehicles

Hongjun Choi
Purdue Univesity
choi293@purdue.edu

Sayali Kate
Purdue Univesity
skate@purdue.edu

Yousra Aafer
University of Waterloo
yaafer@uwaterloo.ca

Xiangyu Zhang
Purdue Univesity
xyzhang@cs.purdue.edu

Dongyan Xu
Purdue Univesity
dxu@cs.purdue.edu

ABSTRACT

We propose a new type of vulnerability for Robotic Vehicles (RVs), called Cyber-Physical Inconsistency. These vulnerabilities target safety checks in RVs (e.g., crash detection). They can be exploited by setting up malicious environment conditions such as placing an obstacle with a certain weight and a certain angle in the RV's trajectory. Once exploited, the safety checks may fail to report real physical accidents or report false alarms (while the RV is still operating normally). Both situations could lead to life-threatening consequences. The root cause of such vulnerabilities is that existing safety checks are mostly using simple range checks implemented in general-purpose programming languages, which are incapable of describing the complex and delicate physical world. We develop a novel technique that requires the interplay of program analysis, vehicle modeling, and search-based testing to identify such vulnerabilities. Our experiment on 4 real-world control software and 8 vehicles including quadrotors, rover, and fixed-wing airplane has discovered 10 real vulnerabilities. Our technique does not have false positives as it only reports when an exploit can be generated.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Computer systems organization → Robotic control**; *Embedded and cyber-physical systems.*

KEYWORDS

CPS Security; Robotic Vehicle; Cyber-Physical Inconsistency

ACM Reference Format:
Hongjun Choi, Sayali Kate, Yousra Aafer, Xiangyu Zhang, and Dongyan Xu. 2020. Cyber-Physical Inconsistency Vulnerability Identification for Safety Checks in Robotic Vehicles. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3372297.3417249

CCS '20, November 9–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3417249

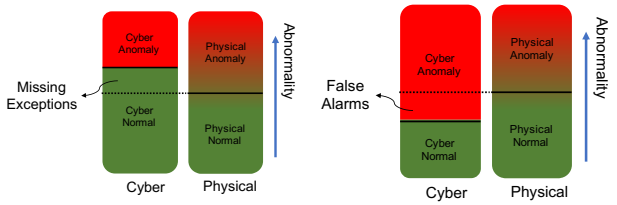
1 INTRODUCTION

Robotic Vehicles (RVs), such as self-driving cars and planes [9, 17, 19, 49, 80], operate autonomously with the harmonious coupling of cyber and physical components. The physical components, e.g., sensing and actuation devices, interact with the physical world, whereas the cyber components, e.g., control software, process sensor signals, make autonomous decisions, and operate the actuation devices. RVs are safety-critical. Misbehavior may likely cause physical damages, some being life-threatening. In order to mitigate the consequences, today's RV control software equips with various *safety checks* to detect any abnormal situation and perform predesigned counter-measures. These checks utilize various sensor and state information to assess the physical condition of the vehicle and the environment, and detect emergent situations. When safety violations are detected, counter-measures will be taken right away to mitigate damages. For examples, modern ground vehicles are often equipped with an airbag system, which is designed to inflate air bags instantly to protect passengers when a collision is detected. Aerial vehicles are often equipped with parachute releasing and emergence landing systems.

The correctness of safety checks are hence of critical importance. These checks are an integral part of control software usually implemented in some general-purpose high level programming language that was not designed to describe complex physics. They are often implemented as a sequence of conditional statements that validate the ranges of certain state/sensor values. However, the simple boolean semantics of conditional statements have limited expressiveness and hence can only approximate the boundary between safe and unsafe states. Accuracy loss in approximation may lead to *over-approximation* vulnerabilities, which are essentially false alarms (e.g., the system reports crashes but there are no real physical crashes), and *under-approximation* vulnerabilities, which are real exceptions that the system fails to detect. We refer to these vulnerabilities as *Cyber-Physical (CP)-inconsistencies*. Such vulnerabilities have catastrophic consequences, as illustrated by many recent accidents. For example, Tesla's autopilot caused a fatal crash with a white trailer in 2016 [75], due to an under-approximation vulnerability of the safety checks; that is, the control program failed to detect the risky situation accurately and did not trigger the counter-measure. More recently, two Boeing-737 Max airplanes crashed in 2018 and 2019 [10, 11], respectively. It was reported that in both accidents, the Maneuvering Characteristics Augmentation System (MCAS) (i.e., anti-stall system), a safety-check component, was improperly activated in the presence of erroneous sensor readings,

denoting an over-approximation vulnerability. These vulnerabilities may be intentionally and systematically exploited by malice (e.g., through creating the failure inducing environmental conditions) to launch attacks that can lead to similar critical physical damage.

Figure 1 illustrates the root causes of CP-inconsistencies. In the left sub-figure, the *Physical* bar denotes the state space in the real physical world, with red representing anomaly and green safety. Observe that since the physical world is continuous and highly complex, the boundary between the two sub-spaces is blurry and gradual. Ideally, the safety function shall be conservative, excluding all possible unsafe states. In the figure, the boundary denoted by the function shall fall into the fully green area. However, it is very difficult for RV system developers to use a set of range checks to describe the highly complex, non-linear, and (maybe) non-convex safe/unsafe state space, due to the complicated correlation among state/sensor variables. As such, the boundary implemented by safety checks in the cyber-space (illustrated in the *Cyber* bar on the left) admits unsafe physical states, leading to under-approximation. The two bars on the right similarly demonstrate over-approximation. As we will illustrate in the motivation section, developers tend to use configuration parameters that are derived from their domain knowledge or even just conjecture in safety checks. For example, they consider an angle deviation larger than 30 degree must indicate a crash. However, there is no rational for this magic number of 30, which may or may not correspond to a real crash, depending on many other environmental factors.



(a) Under-approximation **(b) Over-approximation**

Figure 1: CP-inconsistencies

Three parties are involved in CP-inconsistencies: *control software*, the *vehicle*, and the *environment*. As such, CP-inconsistency detection techniques ought to be able to model and reason about these three aspects. The large body of existing software vulnerability detection techniques are mostly based on program analysis or program testing [16, 18, 21, 23, 54, 66, 69, 73, 82], and hence miss the other two aspects. As such, they may report exploits that are infeasible in the physical world. Recent works on using control invariants to detect attacks on RV systems [15] model the program and vehicle aspects as they leverage vehicle dynamics and program instrumentation to detect deviations. However, they do not systematically consider the *environment* variable. Note that an exploit to a CP-inconsistency vulnerability is essentially a set of realistic environmental conditions under which the RV misbehaves, just like in the Boeing-737 MAX cases. In practice, RVs need to go through substantial physical field tests and crash tests [3, 8, 12, 22] before they are released. For example, Telsa tests self-driving cars on California roads and has to regularly report operation statistics to a state agency. However, such physical tests can hardly stress the software aspect. They are also extremely expensive and unlikely thorough, while CP-inconsistencies are often boundary/corner cases.

This paper proposes a novel testing based *CP-inconsistency* detection technique. In order to cohesively reason about the three aforementioned aspects, it requires the inter-play of program analysis, RV system modeling, and high fidelity environment-aware simulation, orchestrated by a multiple-objective search based test driver. Specifically, program analysis is developed to extract the safety checks, which are a set of predicates over state/sensor variables guarding some kind of safety violation. Both the subject RV dynamics and its control logics are modeled to a list of equations through *System Identification* (SI) [58]. These equations describe how the vehicle behaves given the control objectives (e.g., a reference position) and the current states. SI derives such equations through regression over a set of collected traces of vehicle operation. Note that the SI model does not consider the environment. *One can consider it as a virtual RV (VRV) that operates in an ideal world without any environmental interference.* A *virtual test field* (VTF) is designed to consider all the relevant environmental factors. For example, to test drones, the important factors are wind and obstacles with different physical properties (e.g., shape, weight, or inertia). VTF is like a real physical test field. The difference is that it is realized inside a simulator, allowing a massive number of testings. In a test, both the RV and the VRV operate on the same input, which includes a navigation plan and environment setup. A "real" crash is detected by observing substantial state deviation between the VRV and the RV (in simulated VTF). For example, assume the RV hits an obstacle on the VTF and completely stops (i.e., a crash). The VRV is still moving as planned as it is oblivious to the VTF. An under-approximation CP-inconsistency is identified if such a crash cannot be detected by the safety checks. The test driver models the discrepancy between the values of crash check expressions extracted by program analysis, which measure how close it is to detect the crash, and the real crash to a multiple objective function (over environmental variables). A search algorithm is then used to minimize/maximize the objectives, in order to induce an inconsistency. For example, if the test driver senses that introducing stronger wind and/or reducing the weight of an obstacle helps enlarge the discrepancy, it will continue to do so.

Contribution. Our contributions are summarized as follows.

- We introduce a new type of vulnerability – CP-inconsistency – for RVs, induced by inherent inadequacy of general-purpose high level programming languages in describing complex physics. These vulnerabilities are safety related and can be exploited by solely manipulating environmental conditions. They could lead to (life-threatening) physical damages.
- We propose a novel testing based technique to detect such vulnerabilities. The technique features innovative inter-play among program analysis, RV system modeling, simulation, and multi-objective search based testing. It introduces virtual RV and uses it to expose real crashes. This allows solving a critical challenge - the construction of *test oracle* that provides the ground truth.
- We implement a prototype and apply it to 4 real-world control programs for 8 robotic vehicles, including ground rover and aerial quad-rotors. Our prototype finds 10 real CP-inconsistency cases. These cases lead to disrupt normal operations (e.g., unexpected landing) and deadly crashes.

Threat Model. The attacker does not have access to the internals of the RV system. He exploits an RV system with CP-inconsistency by only manipulating external physical conditions. There are two kinds of exploits. The first induces over-approximation vulnerabilities such that the target RV stops operation (due to safety concerns) while it is still functioning properly. The second induces under-approximation vulnerabilities such that the RV continues to operate while a malfunctioning has occurred. Neither requires exploiting any cyber attack vector.

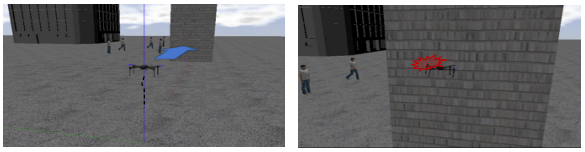
2 MOTIVATION

In this section, we use an example to motivate our technique. The example demonstrates an under-approximation CP-inconsistency vulnerability in ArduCopter [1], one of the most widely used drone control programs. The vulnerability renders a quad-rotor unable to detect physical crashes.

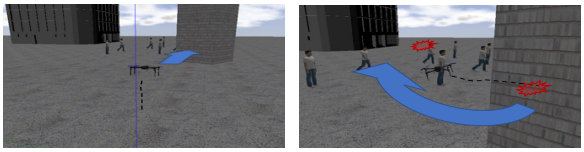
```
1  #define CRASH_CHECK_ACCEL_MAX 3.0f
2  #define CRASH_CHECK_ANGLE_DEVIATION 30.0f
3  #define CRASH_CHECK_TRIGGER_SEC 2
4
5  // conditions for crash check
6    if(!motor->armed() || land_completed) {
7      crash_counter = 0;
8      return;
9    }
10   if(mode == ACRO || mode == FLIP) {
11     crash_counter = 0;
12     return;
13   }
14   if(accel.length() >= CRASH_CHECK_ACCEL_MAX) {
15     crash_counter = 0;
16     return;
17   }
18   if (angle_err <= CRASH_CHECK_ANGLE_DEVIATION) {
19     crash_counter = 0;
20     return;
21   }
22   crash_counter++;
23
24   if (crash_counter >= CRASH_CHECK_TRIGGER_SEC) {
25     gcs().send_text("Crash: Disarming");
26     init_disarm_motors();
27   }
```

Figure 2: Simplified example code of the crash checker

Figure 2 shows a simplified code snippet for crash check in ArduCopter. The code is regularly executed by the scheduler as part of the main control loop with a 400Hz frequency. It determines whether the vehicle encounters a crash. In the code, five conditional statements are used to perform the check. If any of these conditions is satisfied, `crash_counter` is reset to zero, indicating no crash. The first check (line 6) means that in a crash, the vehicle must have the motor armed and it is not landed. The second check (line 10) means that in a crash, the drone must not be in the ACRO or FLIP flight modes, which allow turnover. In the third check (line 14), the acceleration must be smaller than `CRASH_CHECK_ACCEL_MAX` (a pre-defined constant value), indicating the drone is not under control. The fourth (line 18) means that the angle error (between target and current) is larger than `CRASH_CHECK_ANGLE_DEVIATION` for a crash. If all the above conditions are not satisfied in each control loop iteration, `crash_counter` is incremented by one (line 22). Finally, the fifth check (line 24) determines that a crash has occurred when `crash_counter` is greater than or equal to `CRASH_CEHCK_TRIGGER_SEC` (e.g., 2 seconds) and accordingly takes the counter-measure (line 26). Note that by analyzing the program alone, without considering the physical vehicle or the environment, one cannot determine if the system is vulnerable.



(a) Head-on crash and crash detected (disarming)



(b) Side crash and detection failed (subsequent crashes)

Figure 3: Different crash situations and reactions: (a) crash determined, (b) no crash determined. The simulation demos are available at [30, 38])

To understand how the above crash checks lead to a wrong decision during an actual crash, we compare two different crash situations and show how the checks are executed correspondingly. Figure 3 shows the two different crash conditions while the quadrotor performs a mission in the urban area. Figure 3a shows a head-on collision where the quadrotor hits a heavy object with its front. This causes the quadrotor to be stationary after the crash. In contrast, the crash in Figure 3b illustrates a side collision where the quadrotor hits a light object with some angle, and the impact is less than the head-on collision. After the crash, the heading direction is changed by the force caused by the collision and the quadrotor loses control and bounces away in a random direction. Note that both cases have critical impact on the safety of the vehicle and its surroundings (e.g., causing potential physical damages). For instance, the side collision could cause the quadrotor to smash to the ground or into people. Hence, counter-measures – e.g., disarming motors and releasing a parachute – should be taken to prevent subsequent damage.

Back to the code, the checks successfully detect the first situation (i.e., the head-on collision), but miss the second (i.e., the side collision). Specifically, in the second case, the third check takes the true branch (and hence not a crash) because the acceleration is larger than `CRASH_CHECK_ACCEL_MAX` after the crash as the vehicle deflected but did not stop, and even if it had momentarily stopped, the fifth check would take the false branch because `crash_counter` did not exceed `CRASH_CHECK_TRIGGER_SEC`. Observe that the root cause is that the set of pre-defined parameters and the simple range checks of the RV's states are too coarse to describe the delicate and varying physical crash situations. We highly doubt that tuning the parameters would lead to sound and practical solutions due to the highly complex inter-connections among these state/sensor values and between these values and the environment. One could set `CRASH_CHECK_TRIGGER_SEC` to a large value such as 20 seconds, which may never create false alarms. However, it is practically meaningless as with such a long reaction delay, damage has already been incurred. While these problems can be substantially mitigated on traditional manned vehicles by human drivers, they pose prominent challenges for RVs.

Existing RV Testing. RV (physical) destructive test [3, 53] is performed to ensure that safety design meets requirements. For example, RV collision test intentionally crashes the vehicle to analyze its

impact [22, 50]. These tests are usually very expensive and hard to conduct under various situations. Because of the cost of real tests, simulated tests are also performed (e.g., LS-DYNA [59]) to study RV's behaviors in a collision. However, these tests are mostly focusing on the physical aspects. They do not inspect the software and are not able to use program information to improve testing. It is difficult for them to expose corner cases in which CP-inconsistencies mostly reside.

Traditional software testing technique such as fault injection [47] can be conducted for RV control programs. For example, software fault localization (SFL) for control software [46] has been proposed to inject specific types of program bugs (e.g., erroneous arithmetic operations) and then try to locate them through testing. While the technique can be used to determine if the injected bugs actually cause physical misbehaviors, they cannot detect CP-inconsistencies that are caused by latent bugs in control software and require considering the environment. RVFuzzer [55] performs fuzzing to identify program parameters that fall in the specified legal ranges but cause RV mis-behaviors. It monotonically increases/decreases parameter values from the original/default ones until the resulted vehicle operation trace significantly deviates from the trace of the original parameters. In order to exploit the problems reported by RVFuzzer, the attacker needs to be able to replace the default parameters with the problematic ones (e.g., through some social engineering). In contrast, the attack vector of CP-inconsistencies is just environmental conditions. There is no need to access the RV or replace parameters. In our motivating example, the vulnerability can be exploited by setting the weight and the angle of the obstacle.

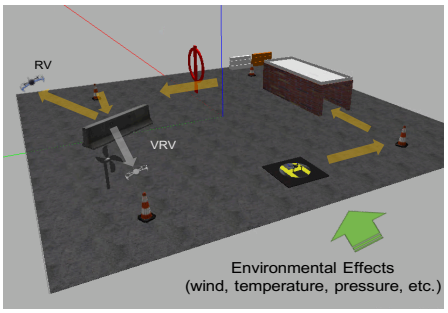


Figure 4: Virtual Test Field

Our approach. To systematically identify CP-inconsistencies, our technique provides a virtual test field (VTF), which can be configured to have various objects and environmental conditions. Figure 4 shows an example VTF. Observe that there are various object models, including wall, pole, box, etc. Their physical properties such as position, rotation, shape, size and inertia are configurable and continuously mutated during testing in order to expose vulnerabilities in the subject RV. The environmental effects include wind, atmosphere (temperature, air pressure), magnetic fields, etc., which are directly related to the specific sensor measurements of the subject RV. In a test, the RV is given a trajectory that traverses the objects on the field (just like in a real physical field test). The test execution is through the Gazebo simulator [62], which is equipped with dynamics engines (e.g., Open Dynamics Engine [61]) to simulate various kinds of RVs and physical objects. We further develop our own plugins to support customized and realistic environmental effects, including dynamic wind direction, duration, and speed. Through system identification (SI), we construct a model for the RV system to describe both the vehicle dynamics and control algorithms. The model can be considered as a virtual RV (VRV) that takes the control reference points (e.g., position and velocity targets) and the current vehicle states, and then produces the next states. During testing, the navigation trajectory is interpreted to a sequence of low level control reference points that are sent to both the RV and the VRV. The RV operates on the VTF, considering the interference from the external physical objects and environmental conditions, whereas the VRV operates in an ideal world, without any external inference (e.g., no wind or obstacles). A crash occurs when substantial state differences are observed between the RV and the VRV. *This is key to our technique because otherwise, we will have to use a set of range checks on sensor/state values of* ***the RV alone*** *to determine if a crash happens, which suffers from the same problems as those existing safety checks.* For instance, in Figure 4, the quadrotor hits a wall and deflects while the virtual quadrotor flies through as it is not interfered by any external factors.

Sound safety checks are supposed to detect the crash when it occurs. To expose a CP-inconsistency vulnerability, our test technique aims to mutate the VTF settings such that a crash can be induced but cannot be detected (i.e., under-approximation). In order to achieve our goal, we define objective functions that consider two kinds of cost, namely *cyber cost* and *physical cost*. The former measures how close the safety checks are to detect the crash and the latter measures the state differences between the RV and the VRV. The cyber cost is constructed by analyzing the control program and collecting the conditions in the predicates that are associated with safety-checking. In our example (Figure 2), the cyber cost includes expression (`CRASH_CHECK_ACCEL_MAX` − `accel.length()`) derived from the third check at line 14, and (`angle_err` − `CRASH_CHECK_ANGLE_DEVIATION`) derived from the fourth check at line 18. Observe that reducing these expressions pushes the predicates from a false value (i.e., crash) towards a true value (i.e., no crash). As will be explained later in Section 4.2, the cyber cost function also models the constraints in other checks. Details are elided at this point. On the other hand, the physical cost is computed as the state differences of the subject RV and the VRV, such as position and velocity differences. As such, exposing under-approximation type of CP-inconsistencies is achieved by using an optimization procedure to mutate the VTF configuration to minimize the cyber cost and maximize the physical cost; conversely over-approximations are exposed by maximizing the cyber cost and minimizing the physical cost. For the example, our technique determines that when the wall with 24kg and inertia $(Ixx, Iyy, Izz) = (12.3, 15.4, 3.0)$ is in the way of the vehicle with a collision angle of 28.6 degrees, the aforementioned under-approximation case can be triggered.

3 OVERVIEW

Figure 5 presents an overview of our system, which consists of two main components: cost function generation and search-based/evolutionary testing. In the upper-left corner, static program analysis is used to identify the predicates guarding the execution of counter-measure

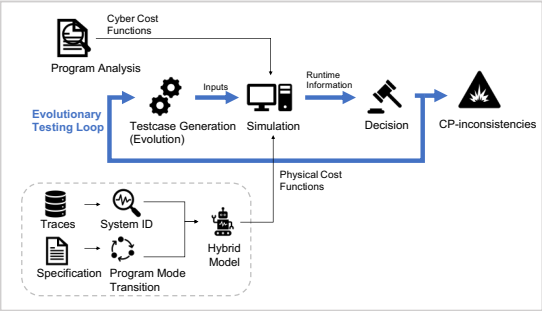


Figure 5: Overview of our test framework

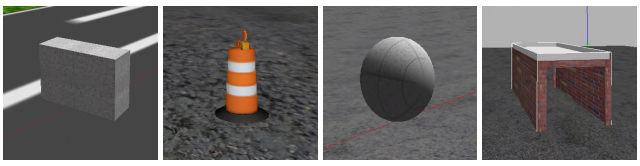
functions such as a parachute releasing function. Currently, such functions are manually provided by the user. They are usually in a small number. The extracted predicates are used to derive the cyber cost functions and a set of constraints (e.g., the quadrotor must not be in a landed mode). In the lower-left corner, system identification (SI) [60] makes use of operation traces (e.g., stage logs) and a model template to derive a *state-space* model for the subject RV. The model describes both the vehicle dynamics and the control algorithm and predicts the next state from the control references and the current state. The template determines the complexity of the generated model (e.g., linear versus non-linear). SI is essentially a procedure to derive the coefficients for the template through regression. A realistic RV system often has various operation modes (e.g., take-off, loiter, and landing). Different modes likely have different models. Hence, our system takes the mode transition specification and represents it as a finite-state machine. The system model is the finite-state machine together with a set of state-space models, one for each operation mode. We also call the system model the *hybrid model*, which is essentially the VRV.

The loop in the middle denotes the search-based/evolutionary testing procedure. For each test case, the RV system is executed in the simulator while the VRV is executed in parallel without external interference. The runtime information of the two executions is collected and compared to see if any CP-inconsistency happens. Specifically, it monitors whether the safety checks fail to report exceptions while physical anomalies actually occur, or vice versa. The evolutionary testing technique generates inputs based on a genetic algorithm for multi-objective optimization. Genetic operations (crossover and mutation) are leveraged to derive new test cases from existing ones, with the guidance of the cost functions.

4 DESIGN

4.1 Simulation Environment

Our technique leverages high-fidelity simulation to reduce the expensive physical testing. Once the simulation based testing discloses CP-inconsistencies, we further reproduce them in the real-world for those that we have the corresponding physical vehicles. Specifically, we use Gazebo [62] for realistic 3D simulation. Gazebo is a popular open-source robotics simulator. It supports various dynamics engines and complex, realistic physical environments (e.g., through the Open Dynamic Engine [70]). It can simulate the dynamics interactions among objects, and between objects and the environment. The simulation has two main configurable components described by the Simulation Description Format (SDF) [63].



(a) Weighted Wall (b) Cylinder Post (c) Rollable Ball (d) Tunnel

Figure 6: Various (different physical properties) example obstacles

SDF is a standard XML format that describes (static and dynamic) objects and the environment for complex simulation, visualization, and control. It mainly consists of two components: *world* and *model*. A *world* consists of objects and a set of environmental parameters (e.g., wind, atmosphere, magnetic field, and light). An object is an instance of some *model* that ranges from simple shapes to complex 3D robots that have different physical properties. In our prototype, we use simple shapes to simulate various kinds of physical obstacles as shown in Figure 6. Note that the different objects have different physical properties (e.g., shape, friction, and elasticity).

In addition to the default environmental effects, we have also implemented additional customized effects as Gazebo *plugins*, which are C++ functions directly accessing Gazebo primitives to provide more physical effects (e.g., wind gust).

4.2 Cost Function Generation

In this section, we present how our technique defines the cost functions systematically. They are used as the guidance for the evolutionary testing. The cost functions consist of cyber and physical objectives, and the testing loop tries to maximize one and minimize the other to identify CP-inconsistencies.

4.2.1 Cyber Cost. The cyber cost function is constructed from the control program through static program analysis. To begin, the user provides a list of counter-measure functions/statements such as the parachute releasing function. The predicates that guard the execution of the counter-measures are extracted and then transformed to a cyber cost function. Specifically, a whole-program *control flow graph* (CFG) is constructed. Nodes in the graph are statements and edges denote control flow. Call relations between functions are represented by edges between a function invocation statement in the caller function and the entrance statement of the callee function. Control dependence can be computed from the whole-program CFG. A statement is control dependent on a particular branch of a predicate if and only if its execution is directly determined by the predicate taking the particular branch. Starting from a counter-measure function-invocation/statement, we acquire the control dependence transitive-closure till the control loop is reached. In other words, all the predicates (and the corresponding branches) guarding the counter-measure and inside the control loop are extracted.

Figure 7 illustrates our analysis using an example. It first constructs a control flow graph (CFG) in (b). Then, it computes post-dominator relations [4] in the CFG and constructs a control dependence graph (CDG) in (c) that captures the control dependences

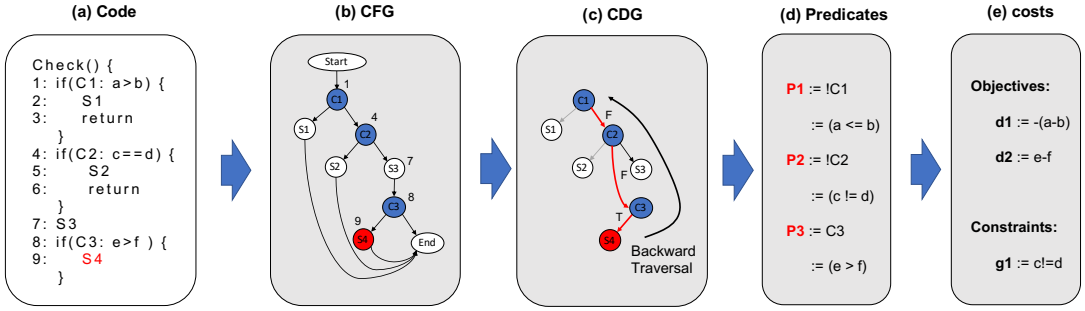


Figure 7: Program analysis and cyber cost function generation

between statements. From the annotated "counter-measure" statement `S4` (in red), it traverses the CDG backward until it reaches the control loop and collects the predicates along the way. In the figure, `S4` is control dependent on $\texttt{C3}^{\texttt{T}}$, meaning `C3` taking the true branch. `C3` is in turn control dependent on $\texttt{C2}^{\texttt{F}}$ as the execution of `C3` is directly determined by `C2` taking the false branch. Finally, predicates and branches $\texttt{C1}^{\texttt{F}}$, $\texttt{C2}^{\texttt{F}}$, and $\texttt{C3}^{\texttt{T}}$ are extracted.

Our analysis is inter-procedural because safety checks may cross multiple functions. It leverages the default points-to analysis in the compiler [72] to handle function pointers. Although irregular control flow such as recursion posts challenges for control dependence computation, we have not encountered such cases in our subject systems. This is reasonable because control loop is time sensitive such that developers avoid putting heavy loops or recursions in it.

Each extracted predicate is then normalized to yielding a true value. That is, a predicate with the false branch is normalized to its negation. It is to simplify the later analysis/transformation. In our example (as shown in Figure 7(d)), $\texttt{C1}^{\texttt{F}}$ is normalized to `a<=b`, $\texttt{C2}^{\texttt{F}}$ is normalized to `c!=d`, and $\texttt{C3}^{\texttt{T}}$ is normalized to its original form. The normalized predicates are classified into two kinds. The first includes all the comparative predicates, called the *objectives*. The second includes all the equality or inequality checks, called the *constraints*. In Figure 7(d), `P1` and `P3` are objectives and `P2` is a constraint.

An objective $LHS \circ RHS$ with $\circ$ being any comparison operation (e.g., $\leq, \geq, >, or <$) is transformed to expressions $+/-(LHS - RHS)$ with the sign + or − depending on the comparison direction (i.e, + for $\geq, >$, - for $\leq, <$). We use symbols `d1`, `d2`, etc., to denote these expressions. In our example, the first normalized predicate `a<=b` is transformed to "`d1`: - (a-b)" as shown in Figure 7(e). To identify under-approximation type of vulnerability, test mutation aims to reduce the values of objective expressions `d1`, `d2`, and so on, while making sure the constraints are satisfied. Intuitively, assume the RV is in an unsafe state and the safety-check function detects it. As such, the extracted predicates and their corresponding branches must satisfy (e.g., $\texttt{C1}^{\texttt{F}}$). The corresponding objective expression (e.g., `d1: -(a-b)`) must be greater than 0. Reducing the value of the objective expression helps pushing the safety-check function to negate its decision so that the crash is missed. False positives can be similarly derived.

Formally, the cyber cost is defined as follows.

$$\begin{aligned} &\mathcal{F}_Y = \{d_1, d_2, \ldots, d_m\} \\ &\text{subject to} : \mathcal{G} = \{g_1, g_2, \ldots, g_n\} \end{aligned} \tag{1}$$

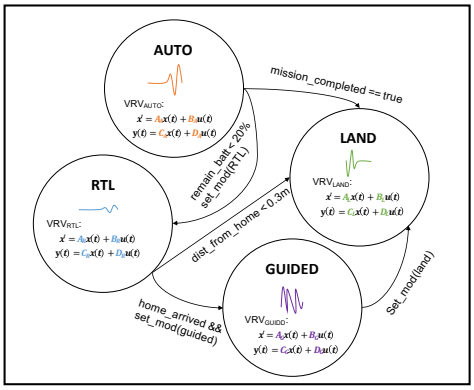


Figure 8: An example of hybrid model: discrete operation mode switch (finite state machine) and continuous state-space models for each operation mode

where $\mathcal{F}_Y$ is the multiple objectives and $\mathcal{G}$ is the constraints.

The cost function for our example is the following form

$$\begin{aligned} &\mathcal{F}_Y = \{d_1 : -(a-b), d_2 : e-f\} \\ &\text{subject to} : \mathcal{G} = \{g_1 : c \neq d\} \end{aligned} \tag{2}$$

.

The cyber cost for the quadrotor safety checks in Figure 2 is as follows.

$$\begin{aligned} &\mathcal{F}_Y = \{d_1 : -(\text{accel\_length} - 3.0), d_2 : \text{angle\_err} - 30, \\ &\quad d_3 : \text{crash\_counter} - 2\} \\ &\text{subject to} : \mathcal{G} = \{g_1 : \text{armed}(), g_2 : \text{!land\_completed}, \\ &\quad g_3 : \text{mode} \neq \text{ACRO}, g_4 : \text{mode} \neq \text{FLIP}\} \end{aligned} \tag{3}$$

4.2.2 Physical Cost. The physical cost function describes how much the current system physical states deviate from the expected states denoted by the virtual RV (VRV). As such, the challenge lies in building the hybrid model (in Figure 8) or the VRV. The hybrid model consists of a finite state machine (FSM) that describes the operation mode (e.g, AUTO and RTL) switches and a list of state-space models, one for each operation mode. The discrete FSM takes the operation mode status as input and performs the corresponding state (operation mode) transition. For instance, the FSM in Figure 8 shows that the mode transition from `AUTO` to `Return-To-Home(RTL)` when the remaining battery is less than 20% and the mode variable is set to `RTL` mode.

The FSM is derived from the RV's operation specification. Most RVs have a small number of operation modes and mode transitions. With our domain knowledge, extracting the needed information

from our subject systems took a few human-hours. As such, the manual efforts of constructing the FSM is reasonable. For each operation mode, system identification (SI) [60] is used to derive a state-space model which describes the continuous physical behaviors. The state-space model can be considered as a set of equations that compute the next state values (e.g., position, velocity, angel velocity, and acceleration), denoted as x' inside the individual modes in Figure 8, from the current states $x(t)$ and a set of reference values $u(t)$, e.g., target positions and velocities. Variable t denotes time. These reference values are generated by the navigation logic of the RV system at a frequency of 400Hz. That is, new targets are provided every 2.5 miliseconds. Note that these reference values are not way-points in the navigation plan which are too coarse-grained. Intuitively, one can consider that the navigation logic continuously interprets the navigation plan based on the current trajectory (or the current deviation from the plan) to a set of smaller goals denoted by reference values, and passes them to the low level control algorithm (e.g., a PID controller), which further interprets them to actuation signals. The state-space model describes the low level control algorithm and the vehicle dynamics. The derivation of state-space model is similar to that in [15, 45, 46, 65] and hence not our contribution. Intuitively, it is done by performing regression on a set of RV operation traces to derive coefficients in a provided model template (e.g., matrices A, B, C and D inside the nodes in Figure 8). Interested readers please refer to [15, 45, 46, 58, 65].

During testing, both the RV and the VRV (i.e., the hybrid model) share the same operation mode inputs and the reference inputs (generated by the navigation logic). The RV will operate on the VTF inside the simulator whereas the VRV simply produces its next states based on the hybrid model. The error between the RV states and the corresponding VRV states constitutes the physical cost. The formal definition is as follows.

$$\begin{aligned} \mathcal{F}_\rho &= \{p_1, p_2, \ldots, p_m\}, \text{ with} \\ p_i &= |\mathrm{RV}_i - \mathrm{VRV}_i| \end{aligned} \tag{4}$$

Here, RV_i and VRV_i denote the ith state of the RV and the VRV, respectively. During testing, $\mathcal{F}_\rho$ is maximized (through input mutation) in order to expose the under-approximation type of vulnerabilities. A weighted sum of p_i is compared to a threshold θ to determine if a real crash happens. If it does and the safety checks cannot detect it, an under-approximation vulnerability is found. We will show in Section 5.2 that our results are not sensitive to θ.

4.3 Multi-objective Evolutionary Testing

Our testing technique continuously alters the test inputs (i.e., VTF configuration) in order to expose CP-inconsistencies. The evolutionary testing guides the input mutation using the cyber and physical costs defined in the previous section. Such guidance is critical given the enormous search space.

4.3.1 Input Selection. The virtual test field (VTF) can be configured in many different ways. Table 1 lists a subset of the variables that can be mutated. As mentioned in Section 4.1, they fall into two kinds, *world* variables that describe the environmental conditions and *model* variables that describe the characteristics of individual objects on the VTF. Depending on the number of objects, the number of variables that can be mutated may be very large. For the motivation example in Section 2, there are 2 types (rotation and inertia) of variables to be mutated for each object. Depending on the subject RV and the kind of safety checks, only a subset of variables have non-trivial influence on system behaviors (and hence on the cost functions). To reduce the search space, we develop an input selection procedure that filters out the insignificant variables for the given cost functions.

Table 1: Input Variables to Mutate

Domain	Input Type	Parameters	Default
World	Wind Direction	[x y z]	0 0 0
	Wind Duration	[Start, End] (s)	0 0
	Wind Strength	N	0
	Gravity	[x y z] (m/s^2)	0 0 -9.8
	Magnetic Field	[x y z] (T)	6e-6 2e-5 -4e-5
	Temperature	T (K)	288.15
	Air Pressure	P (pa)	101325
	Ambient Light	[x y z]	0 0 -1
	Time of Day	[0..24]	10
	Clouds Speed	C (m/s)	0.6
	Fog Density	D	1
Model	Position	[x y z]	0 0 0
	Rotation	[roll pitch yaw]	0 0 0
	Inertia	[ixx iyy izz]	1 1 1
	Mass	m (Kg)	1
	Geometry	Scale	1 1 1
	Bounce	Restitution Coefficient e	0
	Friction	Friction Coefficient μ	1
	Elasticity	Poissons Ratio	0.3

Given a navigation plan and the generated cost functions, we sample each *input variable type* within the valid range uniformly. For each sample, we execute the test in the simulator and observe the cost function differences across samples. The variables whose different samples lead to negligible cost function value differences are pruned out. For example, changing ambient light is less effective when the test is not related to the vision sensor. In contrast, the magnetic field values have substantial impact in the magnetometer-related tests.

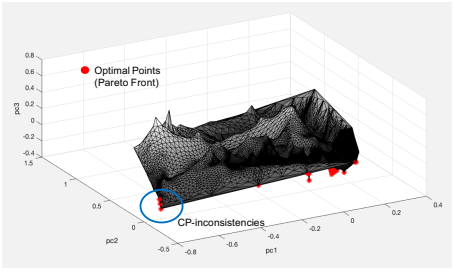


Figure 9: Objective space and optimal set

4.3.2 Evolutionary Testing. The goal of testing is to find maximum/minimum point of the cost functions. However, due to the complexity of physical world and the subject system, the multiple cost functions are unlikely smooth, that is, they are often hilly or even discontinuous. They are typically non-linear and non-convex. Gradient-based optimization techniques [14, 40] are suitable for continuous and convex functions for single objective optimization and hence do not perform well in our case. They often get stuck in local optimals and have difficulties finding the global ones. Also, they do not handle the multiple conflicting objectives well. In Figure 9, we use a very large number of samples 40,000 to approximate

the search space for an IRIS+ RV with the ArduCopter controller. For the purpose of visualization, we use PCA [81] to reduce the high dimensional input space to two dimensions and have the z dimension denote a weighted sum of the costs. Observe that there are lots of ridges and (possible) discontinuities. The red points denote the optimal set (i.e., Pareto optimal [78]) for the multiple objectives, and CP-inconsistencies are located on the bottom left among the optimal points. In our evaluation (Section 5.2), we compare different search algorithms in exposing vulnerabilities.

Therefore, we make use of an evolutionary algorithm instead. Intuitively, the algorithm starts with a set of random test samples that form the first *generation*. It executes these samples in the simulator and computes the cost function values. Those samples that have better cost function values (called *healthy samples*) are selected to derive the next generation. The derivation is through two evolution operations called *cross-over* and *mutation*, with the former mixing parts from two parents into a child input and the latter randomly altering values in a parent input to produce a child. The newly derived children and the healthy parents form the next generation. The process repeats until convergence (no better children can be derived). According to [68], evolutionary algorithms have better capabilities of handling noisy, discontinuous, and even discrete functions. In the following, we explain the details of our solution.

Multi-objective Optimization for CP-inconsistency (MOP-CPi) We reduce CP-inconsistency identification to a Multiple-objective Optimization Problem (MOP) [25, 74]. In the previous section (Section 4.2), we defined two types of cost functions (cyber and physical). Our goal is to search for the maximum difference between the two. In normal conditions, both costs tend to have small values. Conversely, in abnormal conditions, both have large values. Our technique looks for the cases where the difference between the two is maximum such that CP-inconsistency likely occurs. The objective is formalized as follows.

$$\begin{aligned} &\text{minimize } \mathcal{F} = \begin{cases} F_{over} = \{-\mathcal{F}_\gamma, \mathcal{F}_\rho\}, & \text{if } mode = fp \\ F_{under} = \{\mathcal{F}_\gamma, -\mathcal{F}_\rho\}, & \text{otherwise} \end{cases} \\ &\text{subject to } \mathcal{G} = \{g_1, g_2, \ldots, g_n\} \end{aligned} \tag{5}$$

F_{over} represents the objective for identifying over-approximation vulnerabilities (false warnings), where we want to maximize the cyber cost (i.e., minimize the negation of the cyber cost F_γ) and minimize the physical cost F_ρ. F_{under} represents the objective for under-approximation vulnerabilities (missing exceptions), where we want to minimize the cyber cost and maximize the physical cost.

Fitness Ranking Function. Defining a ranking function to measure the level of heath of individual input samples is key to devising an evolutionary algorithm. In our design, ranking is computed based on the aforementioned objective function using (1) the *Pareto optimality level* [25] and (2) the degree of physical anomaly.

Our objective function consists of multiple objectives (e.g., one for each state variable or an expression derived from a safety check). These objectives may contradict with each other, meaning that improving one may undermine others. In our motivation example (an under-approximation vulnerability), the cyber cost (`d1`, `d2`, `d3`) for an input denoting a 0 collision angle is (1, 3, 0). It is (3, 2, 0) when the angle is 30. Observe that the first objective `d1` is better (smaller) with a smaller angle whereas the second objective `d2` is better with a larger angle. Hence, mutating the collision angle input may have contradicting effects for different objectives. *Pareto dominance* [78] was introduced to mitigate such problems. It is widely used in optimization problems in economics, engineering and so on. Intuitively, Pareto dominance states that an input x_1 dominates ($\prec$) x_2 if x_1 is not worse than x_2 in all objectives and x_1 is strictly better than x_2 in at least one objective, which is defined as follows:

$$\begin{aligned} &f_i(x_1) \le f_i(x_2), \forall i = 1, 2, \ldots, M \\ &f_i(x_1) < f_i(x_2), \exists i = 1, 2, \ldots, M \end{aligned} \tag{6}$$

The Pareto optimal set (or *Pareto front*) contains all the elements that are not dominated by others. It is defined as follows.

$$Front(X) = \{x_1 \in X \mid \nexists x_2 \in X \; s.t. \; x_2 \prec x_1 \wedge x_1 \neq x_2\} \tag{7}$$

Based on Pareto dominance, we can determine the optimality level $F(x)$ for each element x. Intuitively, the Pareto set of all elements are at level one. After removing the level one elements, the Pareto set of the remaining elements are at level two, and so on [28].

In addition to the Pareto optimality level, we also consider the physical anomaly level $\Omega(x)$, which is the expected value of normalized physical anomalies. It is defined as follows.

$$\begin{aligned} \omega(x) &= \frac{1}{M_p} \sum_{i=1}^{M_p} \frac{p_i - L_{p_i}}{U_{p_i} - L_{p_i}} \\ \Omega(x) &= \begin{cases} \omega(x), & \text{if } mode = fp \\ -\omega(x), & \text{otherwise} \end{cases} \end{aligned} \tag{8}$$

where M_p denotes the number of physical objectives, U_{p_i}, L_{p_i} upper/lower bounds of objective p_i. Depending on the search mode fp, which is either over-approximation or under-approximation, $\Omega(x)$ is $\omega(x)$ or its negation. Intuitively, we want higher physical anomaly level for under-approximation and lower level for over-approximation. $\Omega(x) \in (0, 1)$ enables selecting the better individuals among those that have the same Pareto optimality level. In other words, the fitness ranking function is the following.

$$R(x) = F(x) + \Omega(x) \tag{9}$$

The individuals with a small R value rank higher and are considered healthier.

Algorithm 1 Evolutionary Testing for CP-inconsistencies Identification

Input: G: number of generation, K: population size
M_c, M_p: number of objectives (cyber,physical)
N: number of inputs
I^L, I^U: input range vector (lower,upper)
Output: CP-inconsistency cases

procedure CPI-Testing
  $X \leftarrow random(I^L, I^U)$ ▷ K input vectors
  $Y \leftarrow evaluation(X)$ ▷ K output vectors
  $P \leftarrow X, Y$ ▷ Initial population of size K
  $R \leftarrow fitness\_rank(P)$ ▷ Fitness Rankings of individuals in P
  for G iterations || CPi found **do** ▷ main evolution loop with K individuals
    $P_{parent} \leftarrow tournament\_selection(P, R, K/2)$ ▷ K/2 parents
    $X_{child} \leftarrow gene\_op(P_{parent}, \mu, \eta)$ ▷ crossover/mutation, K children
    $Y_{child}, R \leftarrow evaluation(X_{child})$ ▷ simulation
    $P_{child} \leftarrow X_{child}, Y_{child}$
    $P_{candi} \leftarrow P_{parent} \cup P_{child}$ ▷ 3K/2 candidates
    $CPi \leftarrow vulnerable(P_{candi})$ ▷ determine CPi cases
    if $|CPi| \neq \emptyset$ **then**
      $save(CPi)$
    end if
    $R_{candi} \leftarrow fitness\_rank(P_{candi})$
    $P, R \leftarrow top\_n\_selection(P_{candi}, R, K)$ ▷ best K individuals
  end for
end procedure

Algorithm 1 presents the details. Each individual is represented as an input vector of size N, with each dimension an input variable. Evaluating an individual produces an output vector of size M, each dimension denoting a cyber/physical objective. The input vector is in the range (I^L, I^U). After each generation, a population includes the best K individuals. In lines 2-3, the algorithm generates random inputs and evaluates them to produce the corresponding outputs. The inputs and outputs together make the initial population P (line 4). Based on the outputs (cyber and physical cost), the fitness ranks are computed at line 5. It then performs evolution over G generations (lines 6-18), or terminates when it converges. The evolution procedure works as follows. First, it performs *binary tournament selection*, which selects half of the population as parents (line 7). Specifically, it picks two individuals randomly and compares their ranks to select the better one. With the $K/2$ parents, genetic operations are used to produce K children (line 8). We use *simulated binary crossover* (SBX) [26] that creates an offspring by combining parts of a pair of parents based on a parameterized probability function which simulates the natural crossover in biology, and *polynomial mutation* [27, 29] that adds a small variation to a parent which simulates biological mutation. The generic operations take the parameters μ and η, which determine how well spread the children will be from their parent in crossover and mutation, respectively. We use a popular setting [28] $\mu = 20\%$ and $\eta = 20$. The children population is obtained with the inputs from the genetic operations and the outputs from `evaluation()` (lines 9-10). Within the current generation, including both the selected parents and the generated children, the `vulnerable()` function determines if the individuals denote CP-inconsistencies. After combining the parents and the children, `fitness_rank()` further computes the rank again and `top_n_selection()` selects the best K individuals for the next generation. Through evolution, the algorithm learns to find the best individuals guided by the cost functions.

5 EVALUATION

We evaluate our technique with four popular RV control programs running on 2 real vehicles and 6 simulated vehicles, including quadrotors, ground rover and fixed-wing airplane.

5.1 Evaluation Setup

5.1.1 Implementation. We have developed a prototype that includes: (1) SDF definitions to describe external environments (world.sdf) and objects (model.sdf) with the SDF versions 1.5 and 1.6 ; (2) Gazebo (version 8.6) plug-ins to generate customized environmental effects based on Gazebo APIs; (3) mission generator in python for System Identification; (4) hybrid model generator implemented on MATLAB; (5) static analysis for cyber-cost function generator; (6) evolutionary test driver that includes a fitness ranking function and genetic mutation operations unique to our technique.

5.1.2 Subject Systems. We evaluate our technique for 7 different types of counter-measure functions. Table 2 shows the control programs and the vehicles. The 6th column (# Modes) represents the number of operation modes such as loiter, auto, rtl, etc. The last column shows the number of safety violations under test. Table 3 shows the types of safety violations that are being checked against

Table 2: Subject Programs

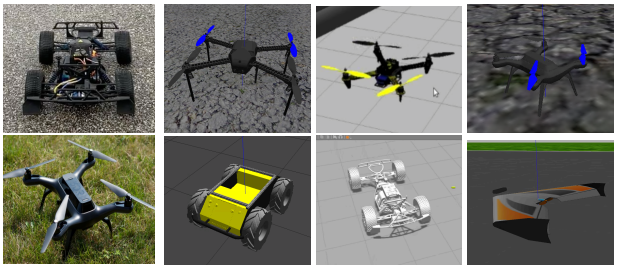
Software	Version	SLOC	Type	Vehicle	# Modes	# Safety violations
ArduCopter	3.7	212,470	Quadrotor	IRIS+	22	4
	3.2	119,326	Quadrotor	Erle-Copter	15	3
	1.5.3	212,601	Quadrotor	Solo	17	4
APMrover2	3.5	203,266	Rover	Husky	12	1
				Erle-Rover	12	1
PX4	1.8.2	346,561	Quadrotor	IRIS+	15	3
				Solo	15	3
ArduPlane	3.10	208,828	Airplane	Zephyr	15	2

Table 3: Subject Safety-check Types

Safety-check Type	ArduCopter	APMrover2	PX4	ArduPlane
Crash	✓	✓	✗	✓
Thrust Loss	✓	✗	✗	✗
Ground Contact	✗	✗	✓	✗
Landing	✓	✗	✓	✗
Freefall	✗	✗	✓	✗
Control Loss (Parachute)	✓	✗	✗	✗
Flying (Landed)	✗	✗	✗	✓

✓: Supported, ✗: Not supported

in each control program. Note that the implementation and predicates for the same type of violation are quite different across control software. For example, in order to check a crash, ArduCopter requires checking the motor status, flight mode, acceleration, angle deviation, etc., while APMrover2 checks the current velocity, motor speed, angular velocity, and so on. Specifically, *Crash* denotes a kind of violation in which the vehicle collides with some object. *Thrust loss* denotes the situation where the thrust of motors is lost, which could lead to severe consequences such as crash. *Landing* denotes the situation where the vehicle is in the process of landing. It is safety related because the navigation logics refer to the status for stable flights. *Freefall* means that the vehicle in free fall. *Control Loss* checks if a vehicle has lost control. For example, in ArduCopter, the vehicle is considered having lost control if it is more than 30 degrees off from the target roll and pitch continuously for at least one second. This leads to disarming motors and releasing a parachute. *Flying* checks if a vehicle is currently flying (or landed). It is safety related because it affects the states of a few critical components (e.g, throttle control).



(a) Real Vehicles **(b) Simulated Vehicles**

Figure 10: Subject Vehicles: (a) Erle-rover, 3DR Solo (b) IRIS+, ErleCopter, 3DR Solo (upper), Husky Rover, Erle-rover, Zephyr (lower)

We used 6 simulated vehicles and 2 real vehicles (3DR Solo and Erle-rover). Gazebo (version 8.6) has high-fidelity phsyics engines and runs virtual vehicles with different environmental conditions. The vehicles are shown in Figure 10. After we identify CP-inconsistency cases, we validate the ones where we have the physical vehicles in the real-world, following the (environmental) exploit inputs generated by our technique.

5.2 Results

We first report the CP-inconsistency cases found by our technique. Then we compare our technique with gradient descent and random search to demonstrate the effectiveness of our technique. Lastly, we show that the identification of these cases is not sensitive to the threshold used in comparing RV and VRV states to detect real accidents (Section 4.2.2).

CP-inconsistencies. Table 4 shows details of the CP-inconsistency cases found by our technique. The information of each case is presented in an upper row and an lower row. The second column and the third column (upper) indicate the controller software and the safety-check type that exhibit the CP-inconsistency. The fourth column (upper) shows the vulnerability type (under- or over-approximation), and the last column (lower) shows the consequences when the vulnerability is exploited. Additionally, VTF configuration, main input type, and input condition columns in the table describe the environmental conditions that trigger the vulnerability. And, #cyber objectives, #constraints and #model states columns provide the number of elements in the cost functions used in the testing that found the vulnerability. The check result column shows the result of the checks (true and false) with the reaction functions after the detection. In ArduCopter, V1 and V2 are the under-approximation (UA) vulnerabilities, as the crash check fails to detect real crashes under the given conditions (check result = false), whereas V3, V4, and V5 are the over-approximation (OA) vulnerabilities, because the checks erroneously detect normal operations as safety-critical situations (check result = true). In PX4, V6 (OA) triggers a false alarm of the freefall check under wind gust, and the landing check of V8 (OA) falsely determines the landing status while the vehicle is hovering under rising wind, whereas V7 (UA) fails to detect actual ground contact. In APMrover2, V9 and V10 (UA) miss the detection of actual crashes. Observe that each case leads to critical and unrecoverable consequences such as crash, drift away, and position lock. We present the actual attacks on real and simulated vehicles as case studies in Section 5.3.

Effectiveness of Search-based Testing. We compare our evolutionary search with gradient descent and random search. In the gradient descent, we approximate the gradients by the variations of inputs over the variation of the weighted sum of multiple objectives. Note that gradients cannot be directly computed using the chain rules [7] as an RV system is not a closed loop. As gradient descent tends to get stuck in local optimals, we run it multiple times and report the best result. We also run a random search, which performs uniform sampling in the same input ranges as those in the evolutionary search. We run it for a number of times that equal to the total number of individuals in the evolutionary search. Table 5 (in Appendix) shows the cost value changes and the testing time with different techniques for all the reported cases. Note that the CP-inconsistency level (IL) indicates the difference between the cyber and physical costs, thus the larger the better for our purpose. Observe that while the evolutionary technique successfully finds the vulnerabilities, the others mostly fail. Specifically, in the evolutionary search, the optimization directions (↓ or ↑) for each cost function (C_n and P) are consistent with the search goals, while the others are not, meaning that other approaches cannot deal with contradictory objectives.

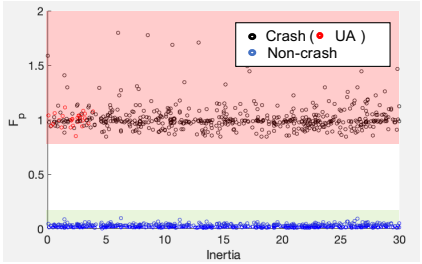


Figure 11: Physical costs and threshold

Physical Accident Detection and Threshold. Recall that our technique uses the physical cost ($\mathcal{F}_\rho$), i.e., the differences between the RV and VRV states, and a threshold to determine a real accident. The detection ought to be precise as it provides the ground truth for the evolutionary search. Figure 11 shows clear distinction between the real crashes confirmed in simulation (red and black circles) and normal executions, which are the blue circles close to the x axis . Each circle denotes the normalized maximum physical cost of a test. Observe that the (pink) area for crashes and the (green) area for none crashes have a big gap. It means that any threshold falling into the gap allows precise detection of real crashes. Moreover, observe that the under-approximation cases (red circles) clearly belong to the pink area, meaning that our method can precisely report real crashes while the safety checks fail.

5.3 Case Studies

In this section, we provide case studies for CP-inconsistency found by our technique. We also discuss how attackers exploit these vulnerabilities to disrupt operations without leaving footprints.

5.3.1 ArduCopter. ArduCopter is equipped with several safety checks. We present a representative over-approximation case in the thrust loss check.

```
#define TLC_ANGLE_DEVIATION_CD 1500
#define TLC_MIN_THROTTLE 0.9f
#define CRASH_CHECK_ANGLE_DEVIATION 30.0f
#define TLC_TRIGGER_SEC 1

if(motors->is_thrust_boost) {
    return;
}
if( (!motors->armed() || land_completed) ||
  (angle_target > TLC_ANGLE_DEVIATION_CD) ||
  (throttle < TLC_MIN_THROTTLE && !motor->limited) ||
  (throttle < 0.25) || (velocity_z >= 0) ||
  (angle_error >= CRASH_CHECK_ANGLE_DEVIATION) {
    thrust_loss_counter = 0;
    return;
}
thrust_loss_counter++;
if(thrust_loss_counter >= TLC_TRIGGER_SEC) { //1(s)
    gcs().send_text("Potential Thrust Loss");
    motors->set_thrust_boost(ture);
}
```

Figure 12: thrust loss check in ArduCopter

Table 4: CP-inconsistencies discovered

#	Controller	Safety-check Type	Vulnerability Type	Mission	# Cyber Objectives	# Constraints	# Model States
		VTF Configuration	Main Input Type	Input Condition	Check Result (Reaction)	Attack Consequence	
V1	ArduCopter	crash check crash with a wall	under-approximation object rotation	straight line [0 0 -28.6]	3 false	5 multiple impact crash	6
V2	ArduCopter	crash check crash with an object	under-approximation weight / inertia	straight line 24 [12.3 15.4 3.0]	3 false	5 multiple impact crash	6
V3	ArduCopter	crash check wind gust	over-approximation wind strength / direction	waypoints (S->N) 16 [-1 0 0]	3 true (disarm)	5 sudden drop	6
V4	ArduCopter	thrust loss check wind gust	over-approximation wind strength / direction / durration	fly up (takeoff) 32 [0 0 -89.9] 1	5 true (thrust boost)	4 sudden acceleration and drift away	6
V5	ArduCopter	parachute check wind gust	over-approximation wind strength / direction	waypoints (W->E) 15->20 [0 1 0]	4 true (parachute)	7 parachute released in normal operation	6
V6	PX4	freefall check wind gust	over-approximation wind strength / duration	hovering 35 / 0.4	3 true (alert)	1 emergency alert in normal operation	6
V7	PX4	ground contact wind gust	under-approximation wind strength / direction / duration	waypoints 31 [0.5 0.3 -0.8] 1.4	3 false	4 bounce and fly away (trajectory deviation)	6
V8	PX4	landing check Rising wind	over-approximation wind strength / duration	hovering 28 / 0.6	4 true (zero thrust)	2 unexpected landing	6
V9	APMrover2	crash check object crash	under-approximation object position	drive forward [5 -0.3 0]	5 false	2 run-off-road multiple crashes	4
V10	APMrover2	crash check non-fixed object crash	under-approximation weight	drive forward 2.3	5 false	2 crashes with a non-fixed object (no response)	4

Thrust Loss. ArduCopter triggers a boost of thrust when it detects thrust loss. Figure 12 shows the simplified code for the thrust loss checks in ArduCopter. In order to identify thrust loss, it has a set of if-conditions (highlighted) on system states. When all these conditions are satisfied to reach the statements at lines 20-21 (i.e., thrust loss is detected for at least `TLC_TRIGGER_SEC`=1 second), the thrust boost function is triggered, which scales the motor thrust by a ratio, to compensate for the detected loss.

To test the thrust loss safety checks, our VTF configuration uses wind gust on the quadrotor flying in the loiter mode, in which the vehicle tries to maintain the target pose - position and attitude. The wind gust causes thrust loss on the vehicle. In this situation, the expected behavior of the checks is to activate the thrust boost so that the vehicle continues to fly against the wind forces.

Our technique was able to find an input condition that causes erroneous detection of thrust loss and the undesired activation of thrust boost (i.e., over-approximation CP-inconsistency). The sudden boost (or acceleration), when there is no real thrust loss, makes the vehicle fly away from the target position. Specifically, the input condition includes the wind gust of speed 32mph in the exact opposite direction of the motor thrust ($< x, y, z >=< 0, 0, -90 >$) for 1 second. Note that under such a short duration of wind forces, the controller can bring the vehicle back to its target pose. However, due to this particular wind gust, all the thrust loss check conditions are satisfied and the motor thrust boost is activated. As the wind gust disappears after 1 second, the motor thrust boost is actually activated under no wind forces. The boost makes all the motors to reach their maximum speed. In practice, the attitude angle (roll and pitch) of the quadrotor during hovering is not perfectly zero all the time. With this small tilt, the maximum motor speed leads the vehicle to move suddenly in an unexpected direction (depending on the attitude angle) leading to a crash or another faulty failure detection (e.g., GPS glitch due to unexpected sudden displacement, even when the GPS sensor is normal).

We tested the case in the real world to confirm that the erroneous behavior actually happens under the reported conditions (wind

(a) Fly with wind gust **(b) Blower and anemometer**

Figure 13: Subject quadrotor and wind gust generator

speed 32mph, direction $< 0, 0, -90 >$, and duration 1 sec). We can inject various wind forces with a 56V leaf blower (with a maximum speed of 125 mph and air volume of 465 cfm), as shown in Figure 13. In this case, the wind gust is turned off after 1 sec, and the thrust boost is not supposed to be triggered. However, it is triggered after the wind gust is off. For the safety issue, we inserted code to change the mode to `Land` immediately after triggering thrust boost function to avoid an accident (i.e., landing instead of full throttle). The video for the erroneous thrust loss check is available at [39].

5.3.2 APMrover2. We present two under-approximation cases for APMrover2. Figure 14 shows the simplified code snippet from the crash checker in APMrover2. It decides whether a crash happens with multiple if-statements (highlighted in the code). If all the conditions hold for more than 2 seconds (defined by `CRASH_CHECK_TRIGGER_SEC`), it determines that the current vehicle is in a `crashed` state, and the controller initiates motor disarming to avoid additional damages. The video at [35] shows a normal crash and detection.

Side Crash into Walls. Our technique reported an under-approximation vulnerability in which the RV misses the actual crash. The rover performs a "line follow" mission in which it drives forward in a straight

```
1  #define CRASH_CHECK_VEL_MIN 0.3      //(m/s)
2  #define CRASH_CHECK_THROTTLE_MIN 5.0 //(%)
3  #define CRASH_CHECK_TRIGGER_SEC 2    //(s)
4
5  if(!is_armed() || !is_autopilot_mode()) {
6      crash_counter = 0;
7      return;
8  }
9  if((abs(pitch) > crash_angle ||   //param:45(default)
10     abs(roll) > crash_angle) {
11     crashed = true;
12 }
13 if(groundspeed() >= CRASH_CHECK_VEL_MIN ||
14   ahrs.get_gyro().z >= CRASH_CHECK_VEL_MIN ||
15   get_throttle() < CRASH_CHECK_THROTTLE_MIN) {
16   crash_counter = 0;
17     return;
18 }
19 crash_counter++;
20 if(crash_counter >= CRASH_CHECK_TRIGGER_SEC) {
21     crashed = true;
22     gcs().send_text("Crash: Going to HOLD");
23     disarm_motor();
24 }
```

Figure 14: Crash check in APMrover2

line and collides with a heavy object (e.g., wall) at location [5, -0.3, 0] with a certain velocity (3m/s) and angle (=23). At the point of the crash, some of the conditions are not satisfied, Specifically, when the pitch and roll angle is smaller than the `crash_angle` threshold, and the `groundspeed` is larger than `CRASH_CHECK_VEL_MIN`, the `crash_counter` is reset, making the crash checker miss the collision, which may deflect the rover and make it out of control.

(a) Erle-rover in a mission **(b) Objects with different weights**

Figure 15: Crash Check with Erle-rover

To validate the vulnerability in the real world, we use hard-board boxes with weight plates inside to realize the reported obstacles (in the simulation). Figure 15 shows the physical test with the weighted obstacles. The vehicle performs "line follows" and hits the "wall" with a 30 degree angle. After the collision, the controller misses the crash. While it is supposed to disarm the motors after the crash, the vehicle continues to drive in a different direction. This causes complete deviation from the expected line (and may cause other damages). The experiment video is available at [36].

Crash into Light Objects. The second under-approximation case may lead to more catastrophic consequences. Specifically, the rover performs the same "line follow" mission and hits a light object (e.g., a kid) head-on this time. After the collision, it continues to move forward since the object's weight is not enough to completely stop the vehicle. In the real experiment, we realize the reported conditions using weighted plates. The object's weight is (=5lbs) , and the rover's velocity is around 3m/s. The object is placed on the expected line. After the crash, it continues to drive forward because the vehicle continues to push the objects slightly, while it is supposed to detect the crash and disarm the motors. Specifically, this situation satisfied the condition at line 13 (i.e., speed > 0.3m/s after crash) and resets the `crash_counter` to zero (i.e., no crash). The video is available at [37].

We have discussion and videos for an over-approximation and an under-approximation for PX4. Due to the space limitations, they are placed in the Appendix.

6 RELATED WORK

Many approaches have been proposed for RV testing in different perspectives [5, 6, 22, 41, 52, 59, 67]. A typical testing approach is live destructive testing with real vehicles under pre-designed scenarios (e.g., crash) and requirements. A crash test is one of the destructive testings. In 2012, Boeing 727 [22] is deliberately crashed for experiments. Many automotive companies perform different types of crash tests (e.g., frontal, side, and rollover) with dummies under government car safety programs (e.g., NCAP [3]). These real vehicle tests can collect the most accurate data from the real crashes, but it is not economical and only performed for certain scenarios. To reduce the cost, virtual tests with simulation, such as LS-DYNA [6, 59, 67], have been utilized. However, the simulation is designed to replace the real vehicles, and certain scenarios (e.g., crash) need to be designed by the engineer. Also, it mostly focuses on the physical impact, and is not control program-oriented.

On the other side, many techniques [20, 42–44, 46, 55, 76] have been proposed to test control programs in RVs. Traditional software testing techniques [2, 21, 57, 64, 82, 83] could find conventional types of bugs or vulnerabilities (e.g., buffer overflow). But our technique handles new types of problems, that is, CP-inconsistency, which is not detected by the traditional approaches. Formal methods [42–44, 76] are effective in exposing various problems in RV software, but have difficulty scaling to whole-system RV verification. A destructive test [46, 55] leverages control semantics in RVs have been recently proposed. [46] uses an SI-derived model for software fault localization (SFL). It intentionally injects program bugs (e.g., wrong arithmetic operations) and then tries to locate these bugs with various testing techniques. [55] intentionally changes parameters in a control program to determine valid ranges. It monotonically increases/decreases parameter values until the vehicle is more deviated than fixed thresholds from the target states. Essentially, these approaches intentionally inject faults into the control program whereas our technique looks for latent problems.

7 CONCLUSION

We present a new type of *Cyber-Physical (CP)-inconsistency* vulnerabilities for RVs. These vulnerabilities can be exploited by merely manipulating the environmental conditions. We propose a novel technique to detect these vulnerabilities. The technique features a novel inter-play of program analysis, RV modeling, and evolutionary search based testing. Our technique detects 10 real vulnerabilities in 4 real-world control software with 8 vehicles.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their constructive comments. This research was supported, in part by NSF 1901242 and 1910300, ONR N000141712045, N000141410468 and N000141712947, IARPA TrojAI W911NF-19-S-0012, Sandia National Lab under award 1701331. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] 2017. Arduino based Arducopter UAV, the open source multi-rotor - Arducopter, the open source UAV multicopter. http://www.arducopter.co.uk/.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 1–40.

[3] National Highway Traffic Safety Administration et al. 2007. The new car assessment program suggested approaches for future program enhancements. *DOT HS* 810 (2007), 698.

[4] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, principles, techniques. *Addison wesley* 7, 8 (1986), 9.

[5] Tejasagar Ambati, KVNS Srikanth, and P Veeraraju. 2012. Simulation of vehicular frontal crash-test. *International Journal of Applied Research in Mechanical Engineering (IJARME) ISSN* (2012), 2231–5950.

[6] Ali O Atahan. 2009. Vehicle crash test simulation of roadside hardware using LS-DYNA: a literature review. *International Journal of Heavy Vehicle Systems* 17, 1 (2009), 52–75.

[7] Robert Gardner Bartle et al. 1976. *The elements of real analysis*. Wiley.

[8] Bloomberg 2017. Tesla Is Testing Self-Driving Cars on California Roads. https://www.bloomberg.com/news/articles/2017-02-01/tesla-is-testing-self-driving-cars-on-california-roads.

[9] Boeing Air Taxi 2019. BoeingâĂŹs Autonomous Taxi Takes Flight. https://www.wsj.com/articles/boeings-autonomous-taxi-takes-flight-11548249580.

[10] Boeing737-Ethiopian 2016. Ethiopian Airlines: 'No survivors' on crashed Boeing 737. https://www.bbc.com/news/world-africa-47513508.

[11] Boeing737-Lion 2016. Lion Air: How could a brand new plane crash? https://www.bbc.com/news/world-asia-46014260.

[12] California DMV 2019. Testing of Autonomous Vehicles with a Driver. https://www.dmv.ca.gov/portal/dmv/detail/vr/autonomous/testing.

[13] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. 2011. Comprehensive Experimental Analyses of Automotive Attack Surfaces.. In *USENIX Security Symposium*. San Francisco.

[14] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.

[15] Hongjun Choi, Wen-Chuan Lee, Yousra Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, Dongyan Xu, and Xinyan Xinyan. 2018. Detecting Attacks Against Robotic Vehicles: A Control Invariant Approach. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 801–816.

[16] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. 2017. Protecting Bare-metal Embedded Systems With Privilege Overlays. In *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 289–303.

[17] CNN 2012. Self-driving cars now legal in California. http://www.cnn.com/2012/09/25/tech/innovation/self-driving-car-california/index.html.

[18] Frederick B Cohen. 1993. Operating system protection through program evolution. *Computers & Security* 12, 6 (1993), 565–584.

[19] comma.ai 2018. commaai/openpilot: open source driving agent. https://github.com/commaai/openpilot.

[20] Mirko Conrad. 2004. *A systematic approach to testing automotive control software*. Technical Report. SAE Technical Paper.

[21] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.. In *USENIX security symposium*, Vol. 98. San Antonio, TX, 63–78.

[22] Crash Test 2012. See What Happens When A Boeing 727 Is Crashed Into The Desert On Purpose. https://www.flightglobal.com/video-boeing-727-deliberately-crashed-in-desert-for-tv/105069.article.

[23] Ang Cui and Salvatore J Stolfo. 2011. Defending embedded systems with software symbiotes. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 358–377.

[24] Drew Davidson, Hao Wu, Robert Jellinek, Vikas Singh, and Thomas Ristenpart. 2016. Controlling UAVs with Sensor Input Spoofing Attacks.. In *WOOT*.

[25] Kalyanmoy Deb. 2001. *Multi-objective optimization using evolutionary algorithms*. Vol. 16. John Wiley & Sons.

[26] Kalyanmoy Deb and Ram Bhushan Agrawal. 1995. Simulated binary crossover for continuous search space. *Complex systems* 9, 2 (1995), 115–148.

[27] Kalyanmoy Deb and Mayank Goyal. 1996. A combined genetic adaptive search (GeneAS) for engineering design. *Computer Science and informatics* 26 (1996), 30–45.

[28] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2 (2002), 182–197.

[29] Kalyanmoy Deb and Santosh Tiwari. 2008. Omni-optimizer: A generic evolutionary algorithm for single and multi-objective optimization. *European Journal of Operational Research* 185, 3 (2008), 1062–1087.

[30] Demo Video 2019. Head-on crash. https://drive.google.com/open?id=1_yHV5eGf13bKISqshB3ieqR62dDYLYgs.

[31] Demo Video 2019. PX4 free fall check. https://drive.google.com/open?id=1pGLktjVZPvGXhugJq02GYxAn67XL7BTY.

[32] Demo Video 2019. PX4 free fall check fail. https://drive.google.com/open?id=19WUb3f_KlUL_Nc8zmwlfKyQysM9KPalG.

[33] Demo Video 2019. PX4 ground contact. https://drive.google.com/open?id=1_T8Lk3FX-ujSxh8nBINRAZUfUoI9sI4g.

[34] Demo Video 2019. PX4 ground contact fail. https://drive.google.com/open?id=14-lausvtfEjFfGJJtKFdEcLG5k2-XDpf.

[35] Demo Video 2019. Rover crash check. https://drive.google.com/open?id=1Wiyz0s8fLZlziB9MFBfriVNBZY8RLyPD.

[36] Demo Video 2019. Rover crash check fail. https://drive.google.com/open?id=1aB2LMoFySZKr0pq_CWOII53qO5URlWXQ.

[37] Demo Video 2019. Rover crash check fail2. https://drive.google.com/open?id=1OnSgODRY2hUwh-9GrCCgKaqk3nzx3Jgn.

[38] Demo Video 2019. Side crash. https://drive.google.com/open?id=1mC-qbZdeWK6ZnHA8RCK-JiTbCnG4jtN8.

[39] Demo Video 2019. Thrust loss check fail. https://drive.google.com/open?id=1odEt6ZMw9nlH7Q7ufXbhD-wHoFzytlbS.

[40] P Dimitri et al. 1999. *Nonlinear programming*. Athena Scientific.

[41] Azim Eskandarian, Dhafer Marzougui, and Nabih E Bedewi. 1997. Finite element model and validation of a surrogate crash test vehicle for impacts with roadside objects. *International Journal of Crashworthiness* 2, 3 (1997), 239–258.

[42] Chuchu Fan, Bolun Qi, Sayan Mitra, and Mahesh Viswanathan. 2017. DryVR: Data-Driven Verification and Compositional Reasoning for Automotive Systems. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 441–461.

[43] Chuchu Fan, Bolun Qi, Sayan Mitra, Mahesh Viswanathan, and Parasara Sridhar Duggirala. 2016. Automatic reachability analysis for nonlinear hybrid models with C2E2. In *International Conference on Computer Aided Verification*. Springer, 531–538.

[44] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. 2011. SpaceEx: Scalable verification of hybrid systems. In *International Conference on Computer Aided Verification*. Springer, 379–395.

[45] Jairo Giraldo, David Urbina, Alvaro Cardenas, Junia Valente, Mustafa Faisal, Justin Ruths, Nils Ole Tippenhauer, Henrik Sandberg, and Richard Candell. 2018. A survey of physics-based attack detection in cyber-physical systems. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 76.

[46] Zhijian He, Yao Chen, Enyan Huang, Qixin Wang, Yu Pei, and Haidong Yuan. 2019. A system identification based Oracle for control-CPS software fault localization. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 116–127.

[47] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. 1997. Fault injection techniques and tools. *Computer* 30, 4 (1997), 75–82.

[48] Todd E Humphreys, Brent M Ledvina, Mark L Psiaki, Brady W O'Hanlon, and Paul M Kintner Jr. 2008. Assessing the spoofing threat: Development of a portable GPS civilian spoofer. In *Proceedings of the ION GNSS international technical meeting of the satellite division*, Vol. 55. 56.

[49] Hyundai S-A1 flying taxi 2019. Hyundai S-A1 flying taxis could take flight by 2023 for Uber elevate. https://www.digitaltrends.com/cars/hyundai-sa1-flying-taxi-ces-2020/.

[50] IIHS 2020. Insurance Institute for Highway Safety (IIHS). https://www.iihs.org.

[51] Rob Millerb Ishtiaq Roufa, Hossen Mustafaa, Sangho Ohb Travis Taylora, Wenyuan Xua, Marco Gruteserb, Wade Trappeb, and Ivan Seskarb. 2010. Security and privacy vulnerabilities of in-car wireless networks: A tire pressure monitoring system case study. In *19th USENIX Security Symposium, Washington DC*. 11–13.

[52] Karen E Jackson, Richard L Boitnott, Edwin L Fasanella, Lisa E Jones, and Karen H Lyle. 2004. A history of full-scale aircraft and rotorcraft crash testing and simulation at NASA Langley Research Center. (2004).

[53] Karim Nice 2001. How Crash Testing Works. https://auto.howstuffworks.com/car-driving-safety/accidents-hazardous-conditions/crash-test1.htm.

[54] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. 2018. Securing Real-Time Microcontroller Systems through Customized Memory View Switching. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium* (San Diego, California) *(NDSS '18)*. The Internet Society.

[55] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. 2019. RVFuzzer: Finding Input Validation Bugs in Robotic Vehicles through Control-Guided Testing. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 425–442. https://www.usenix.org/conference/usenixsecurity19/presentation/kim

[56] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. 2010. Experimental security analysis of a modern automobile. In

Security and Privacy (SP), 2010 IEEE Symposium on. IEEE, 447–462.
[57] Ivan Victor Krsul. 1998. *Software vulnerability analysis.* Purdue University West Lafayette, IN.
[58] L Ljung. 1999. System Identification-Theory for the User, Prentice Hall, Upper Saddle River N. *System identification: Theory for the user. 2nd ed. Prentice Hall, Upper Saddle River, NJ.* (1999).
[59] LS-DYNA 2019. Home | Livermore Software Technology Corp. https://www.lstc.com/.
[60] MATLAB 2017. System Identification Toolbox - MATLAB. https://www.mathworks.com/products/sysid.html.
[61] Open Dynamics Engine 2014. Open Dynamics Engine. https://www.ode.org.
[62] Open Source Robotics Foundation 2014. Gazebo. http://gazebosim.org/.
[63] Open Source Robotics Foundation 2019. SDF Home. http://sdformat.org.
[64] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 75–84.
[65] Fabio Pasqualetti, Florian Dörfler, and Francesco Bullo. 2013. Attack detection and identification in cyber-physical systems. *IEEE Trans. Automat. Control* 58, 11 (2013), 2715–2729.
[66] Lee Pike, Pat Hickey, Trevor Elliott, Eric Mertens, and Aaron Tomb. 2016. Trackos: A security-aware real-time operating system. In *International Conference on Runtime Verification*. Springer, 302–317.
[67] Md Atiqur Rahman and D Praveen Babu. [n.d.]. Simulation of Car Frontal Fascia During Crash using LS-DYNA. ([n. d.]).
[68] Ralf Salomon. 1998. Evolutionary algorithms and gradient search: similarities and differences. *IEEE Transactions on Evolutionary Computation* 2, 2 (1998), 45–55.
[69] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 298–307.
[70] Russell Smith et al. 2005. Open dynamics engine. (2005).
[71] Yunmok Son, Hocheol Shin, Dongkwan Kim, Young-Seok Park, Juhwan Noh, Kibum Choi, Jungwoo Choi, Yongdae Kim, et al. 2015. Rocking Drones with Intentional Sound Noise on Gyroscopic Sensors.. In *USENIX Security Symposium*. 881–896.
[72] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. ACM, 265–266.
[73] Chang-ai Sun, Jingting Jia, Huai Liu, and Xiangyu Zhang. 2018. A Lightweight Program Dependence Based Approach to Concurrent Mutation Analysis. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 116–125.
[74] Hisashi Tamaki, Hajime Kita, and Shigenobu Kobayashi. 1996. Multi-objective optimization by genetic algorithms: A review. In *Proceedings of IEEE international conference on evolutionary computation*. IEEE, 517–522.
[75] Tesla Accident 2016. The technology behind the Tesla crash, explained. https://www.washingtonpost.com/news/the-switch/wp/2016/07/01/the-technology-behind-the-tesla-crash-explained/?noredirect=on&utm_term=.23e1b51bc9e4.
[76] Romain Testylier and Thao Dang. 2013. Nltoolbox: A library for reachability computation of nonlinear dynamical systems. In *Automated Technology for Verification and Analysis*. Springer, 469–473.
[77] Nils Ole Tippenhauer, Christina Pöpper, Kasper Bonne Rasmussen, and Srdjan Capkun. 2011. On the requirements for successful GPS spoofing attacks. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 75–86.
[78] David A Van Veldhuizen and Gary B Lamont. 1998. Evolutionary computation and convergence to a pareto front. In *Late breaking papers at the genetic programming 1998 conference*. 221–228.
[79] Jon S Warner and Roger G Johnston. 2002. A simple demonstration that the global positioning system (GPS) is vulnerable to spoofing. *Journal of Security Administration* 25, 2 (2002), 19–27.
[80] Waymo 2017. Waymo (formerly the Google self-driving car project). https://waymo.com.
[81] Svante Wold, Kim Esbensen, and Paul Geladi. 1987. Principal component analysis. *Chemometrics and intelligent laboratory systems* 2, 1-3 (1987), 37–52.
[82] Zhaogui Xu, Shiqing Ma, Xiangyu Zhang, Shuofei Zhu, and Baowen Xu. 2018. Debugging with intelligence via probabilistic inference. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 1171–1181.
[83] Sai Zhang, David Saff, Yingyi Bu, and Michael D Ernst. 2011. Combined static and dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 353–363.

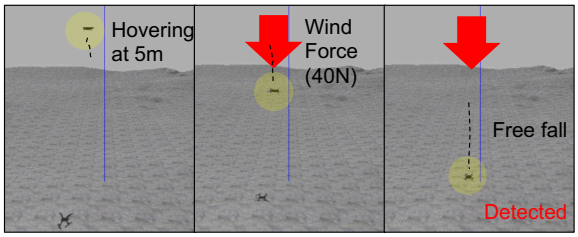
8 APPENDIX

A Additional Case Studies

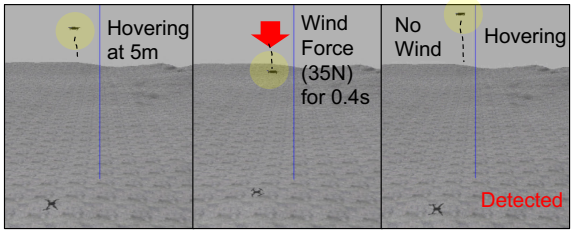
PX4 is another popular control software for quadrotor. It has a number of safety checks such as freefall detection, ground contact detection and landing detection. We present an over-approximation case of freefall detection and an under-approximation in ground contact detection.

```
1 #define LNDMC_FFALL_THR 5.0 //(m/s^2)
2 #define LNDMC_FFALL_TTRI 0.3 //(s)
3
4 freefall_acc_threshold = LNDMC_FFALL_THR;
5 if (freefall_acc_threshold < 0.1f || //parameter check
6     freefall_acc_threshold > 10.0f) {
7     freefallDetected = false;
8 }
9 if (_sensors.timestamp == 0) {  // sensor check
10    freefallDetected = false;
11 }
12 ...
13 if (acc_norm < freefall_acc_threshold) {
14    freefallDetected = true;
15 }
16 if (hysteresis(freefallDetected, LNDMC_FFALL_TTRI))
17    mavlink_and_console_log_info("Freefall detected");
18    ...
19 }
```

Figure 16: Freefall detection in PX4



(a) True positive case (strong wind and free fall/crash)



(b) False positive case (instant wind and continue hovering)

Figure 17: Freefall detection cases (simulation snapshot): PX4 detects both cases as free fall

Free Fall. When PX4 detects free-fall, it emits an emergency message and takes an optional counter-measure such as parachute releasing. Figure 16 shows the simplified code of the freefall checks. To determine the free-fall status, it has several if-conditions (highlighted), including a check against the `freefall_acc_threshold` (=5.0) at line 13. If the `freefallDetected` is true for more than `LNDMC_FFALL_TTRI` seconds (=0.3), the RV decides that the free-fall status is true. Note that the parameter values of `LNDMC_FFALL_THR` and `LNDMC_FFALL_TTRI` are valid within the suggested range.

Figure 17a shows that the free fall check correctly detects the case. Specifically, the vehicle hovers at an altitude of 5 meters (the left), and we generate strong wind forces (=40N) for a few seconds

to simulate the free fall (the middle). The vehicle accelerates toward the ground (the right). During the free fall, the check detects it and reports an emergency message. The video is available at [31].

Figure 17b shows the CP-inconsistency case that our technique found. The check results in an over-approximation (false alarm) under specific environmental conditions. In the same hovering mission (the left), a particular strength of wind force (=35N) directed downward for 0.4 seconds instantly pushes the vehicle and lowers its altitude by around 2 meters (i.e, an instant bounce) (the middle). After the wind is off, the vehicle recovers to the target altitude (=5m) gradually (the right). However, the check detects this case as free fall and triggers an erroneous reaction (e.g., alert and release parachute). Note that this false alarm reaction can interfere with the normal operation. The video for the case is available at [32].

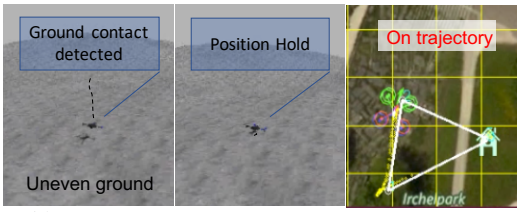
```
1  #define MPC_LAND_SPEED 0.7 //(m/s)
2  #define LNDMC_Z_VEL_MAX 0.5 //(m/s)
3  #define LNDMC_XY_VEL_MAX 1.5 // m/s)
4  #define GROUND_CONTACT_TRIGGER_TIME 0.35 //(s)
5
6  float landSpd = MPC_LAND_SPEED;
7  float maxClimbRate = LNDMC_Z_VEL_MAX;
8  float maxVel = LNDMC_XY_VEL_MAX;
9  float land_spd_thresh = 0.9f * max(landSpd,0.1f);
10 ...
11 if (!_arming.armed)
12     return true;
13 // not in descend and vertical movement
14 if(!_has_low_thrust()
15         && (!_is_climb_rate_enabled()
16             || !(setpoint.vz >= land_spd_thresh))
17         && (abs(_localPos.z_deriv) > maxClimbRate))
18     return false;
19 // horizontal movement
20 if((sqrtf(sq(vx)+sq(vy)) > MaxVel)
21         && _has_position_lock())
22     return false;
23 // vertical movement
24 if(abs(_localPos.z_deriv) > maxClimbRate
25         && _has_altitude_lock())
26     return false;
27
28 return true;
```

Figure 18: Ground contact detection in PX4

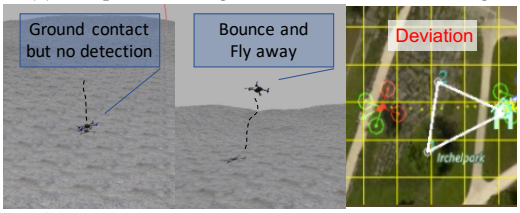
Ground Contact. PX4 checks whether the vehicle hits ground. Specifically, it checks the low thrust or z velocity setpoint (lines 14-17) and no vertical/horizontal movement (lines 20-25). If all the conditions (highlighted in Figure 18) are false for more than `GROUND_CONTACT_TRIGGER_TIME` (=0.35 seconds), ground contact is decided. If it is detected, the position controller turns off the thrust setpoint in body x and y (preventing movement that is not vertical)

Figure 19a shows that the check correctly detects the case. Specifically, the vehicle performs a planned (triangular trajectory) mission and contacts the ground during the flight (the left). The check detects it and lets the controller stop moving along x and y directions to avoid additional physical damage (the middle). In the trajectory view (the right), the vehicle stops on the trajectory safely. The complete simulation video is available at [33].

Figure 19b shows the CP-inconsistency case, where the vehicle hits the ground, but the check fails to detect it. The vehicle performs the same waypoint mission. This is exploited by a wind gust with direction $< 0.5, 0.3, -0.8 >$ for 1.4 seconds and force 31N. The vehicle hits the ground and then completely deviates from the planed trajectory, while it is supposed to detect the ground contact event and the position controller holds its horizontal position. The instant impact of the hit causes a vertical bounce of the vehicle, which causes the z velocity to exceed a threshold and hence fails the detection. The video is available at [34].



(a) True positive case (ground contact and safe holding)



(b) False negative case (ground contact and fly away)

Figure 19: Ground contact detection cases (simulation snapshot)

B Discussion

Countermeasures. The root cause of Cyber-Physical inconsistencies is the incapability of range checks implemented in general-purpose programming languages in describing the physical world, which is continuous and complex. We foresee that a continuous model of the RV expected behavior, like our VRV, shall be a first-order object in the RV runtime that can be queried and compared during safety checks. As such, the runtime system with the proposed solution replaces the incomplete range checks to improve the robustness of future RV systems. This requires enhancing the programming support and the runtime support of RV control software.

External Attacks. The attacker exploits CP-inconsistency vulnerabilities by only manipulating external physical conditions. Our attack vector is realistic and aligned with recently reported CPS attacks on various sensors such as gyroscope, accelerometer, optical flow, wheel speed sensor [13, 24, 48, 51, 56, 71, 77, 79]. These can create adversarial conditions artificially through special devices and setup. In this paper, we demonstrate the feasibility of leveraging small-scale devices (such as wind blowers and weighted boxes) to realistically exploit the reported cp-inconsistencies. Note that more sophisticated devices (e.g., large-scale gust generator and long-range sonic weapons) can be leveraged to generate realistic adversarial conditions. For the cases where we used obstacles, the scenarios are more reasonable if the vehicles are running in an urban area. An attacker can carefully place physical obstacles (e.g., cars) in an RV's trajectory to trigger potential vulnerabilities.

False Negative/Positives. The search space is enormous due to the inter-play of various conditions. Since exploring the search space requires simulation, much more expensive than running a program (like in fuzzing), it is very difficult to establish the ground-truth (and measure FNs). Besides, due to the nature of the genetic algorithm, it is hard to guarantee that we can find all the potentially existing vulnerable cases (i.e., no false negative). Regarding false-positives, our technique is based on high-fidelity simulation. A vulnerability is only reported after it is confirmed. Hence, we don't have false-positives.

Table 5: Comparing search techniques

# (Goal)	Cost Functions	Evolutionary					Gradient Descent					Random Search				
		Cost Change			Time (hr)	S/F	Cost Change			Time (hr)	S/F	Cost Change			Time (hr)	S/F
		B	A	D			B	A	D			B	A	D		
V1 (C↓ P↑)	C1	0.27	0.07	-0.2 (↓)	2.3	S	0.27	0.34	0.07 (↑)	1.9	F	0.27	0.09	-0.18 (↓)	2.3	F
	C2	23.04	17.46	-5.58 (↓)			23.03	37.9	14.87 (↑)			23.04	22.9	-0.14 (↓)		
	C3	0	-708	-708 (↓)			0	-800	-800 (↓)			0	-130	-130 (↓)		
	P	0.98	1.53	0.55 (↑)			0.97	1.23	0.26 (↑)			0.98	1.02	0.04 (↑)		
	IL	0.69	1.62	0.93			0.68	1.10	0.42			0.69	0.80	0.11		
V2 (C↓ P↑)	C1	0.23	0.16	-0.07 (↓)	1.8	S	0.24	0.33	0.09 (↑)	1.6	F	0.23	0.12	-0.11 (↓)	1.8	F
	C2	30.3	27.3	-3 (↓)			30.3	32.8	2.5 (↑)			30.3	23.9	-6.4 (↓)		
	C3	0	-687	-687 (↓)			0	-800	-800 (↓)			0	-85	-85 (↓)		
	P	0.87	1.04	0.17 (↑)			0.87	0.95	0.08 (↑)			0.87	0.9	0.03 (↑)		
	IL	0.5	1.0	0.49			0.5	0.88	0.37			0.05	0.65	0.15		
V3 (C↑ P↓)	C1	-0.2	0.23	0.43 (↑)	3.3	S	-0.2	-3.1	-2.9 (↓)	1.7	F	-0.2	0.41	0.61 (↑)	3.3	F
	C2	-5.5	0.2	5.7 (↑)			-5.4	-6.3	-0.9 (↓)			-5.5	-6.2	-0.7 (↓)		
	C3	-499	0	499 (↑)			-499	0	499 (↑)			-499	0	499 (↑)		
	P	0.08	0.06	-0.02 (↓)			0.07	0.03	-0.04 (↓)			0.08	0.06	-0.02 (↓)		
	IL	-0.08	0.16	0.25			-0.06	-0.09	-0.03			-0.08	0.1	0.18		
V4 (C↑ P↓)	C1	202.63	213.6	10.97 (↑)	2.7	S	202.6	201.6	-1.0 (↓)	2.3	F	202.63	192.2	-10.43 (↓)	2.7	F
	C2	0.02	0.03	0.01 (↑)			0.02	0.09	0.07 (↑)			0.02	0.03	0.01 (↑)		
	C3	0.67	0.68	0.01 (↑)			0.67	0.68	0.01 (↑)			0.67	0.66	-0.01 (↓)		
	C4	28.3	29.1	0.8 (↑)			28.3	28.5	0.2 (↑)			28.3	28.8	0.5 (↑)		
	C5	-400	0	400 (↑)			-400	0	400 (↑)			-400	-400	0 (-)		
	P	0.1	0.08	-0.02 (↓)			0.1	0.09	-0.01 (↓)			0.1	0.06	-0.04 (↓)		
	IL	0.6	0.85	0.25			0.6	0.84	0.24			0.6	0.62	0.02		
V5 (C↑ P↓)	C1	5.2	5.3	0.1 (↑)	3.1	S	5.2	5.5	0.3 (↑)	2.8	F	5.2	5.3	0.1 (↑)	3.1	F
	C2	-8.3	1.5	9.8 (↑)			-8.3	-13.2	-4.9 (↓)			-8.3	-5.2	3.1 (↑)		
	C3	0	-2.3	-2.3 (↓)			0	0.8	0.8 (↑)			0	-0.1	-0.1 (↓)		
	C4	-400	0	400 (↑)			-400	0	400 (↑)			-400	-400	0 (-)		
	P	0.25	0.23	-0.02 (↓)			0.25	0.23	-0.02 (↓)			0.25	0.21	-0.04 (↓)		
	IL	-0.44	-0.14	0.29			-0.44	-0.18	0.25			-0.43	-0.37	0.06		
V6 (C↑ P↓)	C1	-2.0	-2.0	0 (-)	2.9	S	-2	-2	0 (-)	1.3	S	-2.0	-2.0	0 (-)	2.9	S
	C2	7.0	7.0	0 (-)			7.0	7.0	0 (-)			7.0	7.0	(-)		
	C3	-4.8	4.9	9.7 (↑)			-4.8	4.91	9.71 (↑)			-4.8	3.7	8.5 (↑)		
	P	0.07	0.08	0.01 (↑)			0.07	0.08	0.01 (↑)			0.07	0.08	0.01 (↑)		
	IL	-0.82	-0.19	0.63			-0.82	-0.18	0.64			-0.82	-0.26	0.56		
V7 (C↓ P↑)	C1	12.5	8.3	-4.2 (↓)	4.6	S	12.5	13.1	0.6 (↑)	2.1	F	12.5	10.1	-2.4 (↓)	4.6	F
	C2	4.8	0.1	-4.7 (↓)			4.8	0.01	-4.79 (↓)			4.8	0.4	-4.4 (↓)		
	C3	-0.28	0.02	0.3 (↑)			-0.28	0.31	0.59 (↑)			-0.28	0.02	0.3 (↑)		
	P	0.92	0.93	0.01 (↑)			0.92	0.94	0.02 (↑)			0.92	0.94	0.02 (↑)		
	IL	0.41	0.69	0.27			0.41	0.42	0.01			0.41	0.63	0.21		
V8 (C↑ P↓)	C1	0.349	0.348	-0.001 (↓)	2.7	S	0.349	0.348	-0.001 (↓)	1.5	S	0.349	0.348	-0.001 (↓)	2.7	F
	C2	0.349	0.351	0.002 (↑)			0.349	0.348	-0.001 (↓)			0.349	0.348	-0.002 (↓)		
	C3	0.349	0.349	0 (-)			0.349	0.348	-0.001 (↓)			0.349	0.347	-0.001 (↓)		
	C4	-49.2	0.01	49.21 (↑)			-49.2	-72.9	122.1 (↑)			-49.2	24.3	73.5 (↑)		
	C5	-8	0	8 (↑)			-8	-8	0 (-)			-8	-8	0 (-)		
	P	0.08	0.13	0.05 (↑)			0.08	0.09	0.01 (↑)			0.08	0.90	0.01 (↑)		
	IL	0.22	0.47	0.25			0.22	0.45	0.23			0.22	0.35	0.13		
V9 (C↓ P↑)	C1	-38	-39.9	-1.9 (↓)	1.6	S	-38	-39.2	-1.2 (↓)	1.3	F	-38	-37.3	0.7 (↑)	1.6	F
	C2	0.04	-2.14	-2.18 (↓)			0.04	-1.7	-1.74 (↓)			0.04	0.02	-0.02 (↓)		
	C3	0.06	-1.58	-1.64 (↓)			0.06	0.09	0.03 (↑)			0.06	0.16	0.1 (↑)		
	C4	45.0	45.0	0 (-)			45	45	0 (↑)			45	45	0 (-)		
	C5	0	-8	-8 (↓)			0	-8	-8 (↓)			0	-20	-20 (↓)		
	P	0.81	0.79	-0.02 (↓)			0.81	0.88	0.07 (↑)			0.81	0.79	-0.02 (↓)		
	IL	-0.82	-0.5	0.32			-0.82	-0.55	0.26			-0.82	-0.65	0.17		
V10 (C↓ P↑)	C1	-42	-42	0 (-)	2.5	S	-42	-42	0 (-)	1.5	F	-42	-41.7	0.3 (↑)	2.5	S
	C2	0.05	0.02	-0.03 (↓)			0.05	0.07	0.02 (↑)			0.05	0.04	-0.01 (↓)		
	C3	0.08	0.07	-0.01 (↓)			0.08	0.03	-0.05 (↓)			0.08	0.04	-0.04 (↓)		
	C4	45.0	45.0	0 (-)			45	45	0 (-)			45	45	0 (-)		
	C5	0	-20	-20 (↓)			0	-20	-20 (↓)			0	-20	-20 (↓)		
	P	0.8	0.76	-0.04 (↓)			0.8	0.78	-0.02 (↓)			0.8	0.76	-0.04 (↓)		
	IL	-1.13	-0.87	0.26			-1.13	-0.88	0.25			-1.13	-0.85	0.28		

C_n: cyber costs, P: physical cost, B: cost before searching, A: cost after searching,
D: difference (direction), IL: CP-inconsistency level, $(C - P)$ if OA; $(P - C)$ if UA, C is the normalized average of C_n)
S/F: success / fail to find the vulnerability