SpectAcle: Fault Localisation of AI-Enabled CPS by Exploiting Sequences of DNN Controller Inferences

DEYUN LYU and ZHENYA ZHANG, Kyushu University, Fukuoka, Japan
PAOLO ARCAINI, National Institute of Informatics, Tokyo, Japan
XIAO-YI ZHANG, University of Science and Technology Beijing, Beijing, China
FUYUKI ISHIKAWA, National Institute of Informatics, Tokyo, Japan
JIANJUN ZHAO, Kyushu University, Fukuoka, Japan

Cyber-physical systems (CPSs) are increasingly adopting *deep neural networks (DNNs)* as controllers, giving birth to *AI-enabled CPSs*. Despite their advantages, many concerns arise about the safety of DNN controllers. Numerous efforts have been made to detect system executions that violate safety specifications; however, once a violation is detected, to fix the issue, it is necessary to localise the parameters of the DNN controller responsible for the wrong decisions leading to the violation. This is particularly challenging, as it requires to consider a sequence of control decisions, rather than a single one, preceding the violation. To tackle this problem, we propose SpectAcle, that can localise the faulty parameters in DNN controllers. SpectAcle considers the DNN inferences preceding the specification violation and uses *forward impact* to determine the DNN parameters that are more relevant to the DNN outputs. Then, it identifies which of these parameters are responsible for the specification violation, by adapting classic suspiciousness metrics. Moreover, we propose two versions of SpectAcle, that consider differently the timestamps that precede the specification violation. We experimentally evaluate the effectiveness of SpectAcle on 6,067 faulty benchmarks, spanning over different application domains. The results show that SpectAcle can detect most of the faults.

CCS Concepts: • **Software and its engineering → Software testing and debugging**;

Additional Key Words and Phrases: fault localisation, neural network controllers, cyber-physical systems

ACM Reference format:
Deyun Lyu, Zhenya Zhang, Paolo Arcaini, Xiao-Yi Zhang, Fuyuki Ishikawa, and Jianjun Zhao. 2025. SpectAcle: Fault Localisation of AI-Enabled CPS by Exploiting Sequences of DNN Controller Inferences. *ACM Trans. Softw. Eng. Methodol.* 34, 4, Article 110 (April 2025), 35 pages.
https://doi.org/10.1145/3705307

P. Arcaini and F. Ishikawa are supported by Engineerable AI Techniques for Practical Applications of High-Quality Machine Learning-based Systems Project (Grant Number JPMJMI20B8), JST-Mirai.
Authors' Contact Information: Deyun Lyu (corresponding author), Kyushu University, Fukuoka, Japan; e-mail: lyu.deyun.107@s.kyushu-u.ac.jp; Zhenya Zhang, Kyushu University, Fukuoka, Japan; e-mail: zhang@ait.kyushu-u.ac.jp; Paolo Arcaini, National Institute of Informatics, Tokyo, Japan; e-mail: arcaini@nii.ac.jp; Xiao-Yi Zhang, University of Science and Technology Beijing, Beijing, China; e-mail: xiaoyi@ustb.edu.cn; Fuyuki Ishikawa, National Institute of Informatics, Tokyo, Japan; e-mail: f-ishikawa@nii.ac.jp; Jianjun Zhao, Kyushu University, Fukuoka, Japan; e-mail: zhao@ait.kyushu-u.ac.jp.

ACM 1557-7392/2025/4-ART110
https://doi.org/10.1145/3705307

1 Introduction

A **Cyber-Physical System (CPS)** is given by the combination of a physical system (*plant*) and a computational component (*controller*), where the controller monitors and controls the plant according to the physical states of the plant and external environments. Recently, **Deep Neural Networks (DNNs)** have been increasingly used as controllers to perform complex control tasks (e.g., *self-driving* [8] and *robotic arms control* [20]); indeed, they have shown to be able to handle intricate environments more efficiently than classic controllers. Such CPSs driven by AI-based controllers are called as *AI-enabled CPSs* [22, 24, 25, 40, 41, 50, 51, 60].

Despite the great potential in terms of control, the safety of AI-enabled CPSs becomes a major concern. Indeed, a DNN controller may produce unforeseen and inexplicable behaviours that lead the plant to malfunctions and catastrophic hazards, which can further bring irreversible social and economic losses. To avoid these dangerous situations, it is necessary to detect the potential unsafe behaviours of DNN controllers and identify the root causes that produce them.

There have been extensive studies on generating unsafe system executions for these systems, i.e., finding a *time-variant* external input signal from the environment that leads the system execution to violating a temporal-logic specification, mainly from the testing and falsification communities [13, 16, 27, 43, 48, 58, 60]. However, these works mostly demonstrate the presence of unsafe behaviours, without delving into a comprehensive investigation of these behaviours. To assist system engineers in enhancing system safety, it is necessary to identify the *root cause* of the violation: since the decision logic of the AI-enabled system is embedded in the DNN controller, this boils down to localising the faults in the DNN, specifically in the configuration of its parameters (e.g., weights).

However, precisely identifying the model parameters of DNN controllers that are responsible for the violations is a challenging problem. This is due to the unexplainable programming logic of DNNs, which makes it difficult to properly assess which parameters inside a DNN controller are responsible for a given (wrong) decision. Some *fault localisation* approaches have been proposed for DNN classifiers [10, 14, 15, 47, 53]; such approaches consider different inferences of a DNN and compare the outcome with the *ground truth* (e.g., the expected label in classification). However, such approaches are not applicable for fault localisation of DNN controllers, due to *two main issues*:

(1) There is no explicit ground truth for the behaviours of DNN controllers, and the correctness of their behaviours can only be assessed by considering the system level specification. Specifically, this requires executing the whole system controlled by the DNN controller for a time period, and then checking whether the system outputs satisfy the temporal-logic specification.
(2) In an AI-enabled CPS, a system violation is due to a *sequence of control decisions* of the DNN controller, i.e., a sequence of inferences. Hence, fault localisation must analyse sequences of DNN inferences, rather than single ones as done by state-of-the-art fault localisation approaches in the context of other domains such as image classification.

Contributions. In this article, we propose **SPECTrum-based fault localisation for Ai ControlLErs (SpectAcle)**, a fault localisation approach for AI-enabled CPSs. Given a *test suite* (i.e., a set of input signals) and a *system specification*, SpectAcle executes each test in the test suite and obtains: (1) in case of a failing test, the sequence of DNN controller inferences preceding the start of the *violation episode* (i.e., the timestamp when the specification is shown to be violated), (2) the whole sequence of inferences for a passing test. For each inference in each test, SpectAcle assesses the *relevance* of DNN weights on the control decision by calculating their *forward impacts*. It then constructs the *execution spectrum* for each DNN weight, based on the status of test executions (i.e., passed or failed) and relevance of the DNN weight to the control decisions. Finally, it adapts

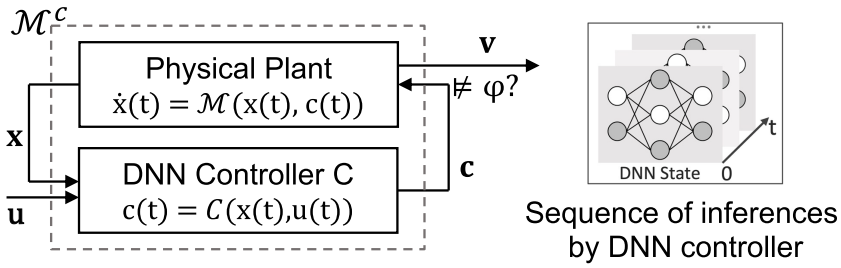

Fig. 1. 1. The architecture of a typical AI-enabled CPS.

existing suspiciousness metrics from SBFL [57] to identify the DNN weights that are more likely responsible for the specification violation.

We propose two versions of SpectAcle, distinguished by the significance assigned to the DNN inferences that may account for the violation, based on their distances to the beginning of specification violation: (1) *unweighted* SpectAcle (SpectAcle$_{uw}$) considers equally all the DNN inferences before the violation; (2) *weighted* SpectAcle (SpectAcle$_{w}$) considers more relevant the inferences closer to the violation, and so weights them more in the construction of the execution spectrum.

We experimentally evaluate the effectiveness of SpectAcle on 6,067 faulty benchmarks, obtained by injecting artificial faults into the DNN weights of 13 correct AI-enabled CPSs taken from existing literature [43, 48, 51, 60]. The results show that SpectAcle is able to report most of the artificial faults, which demonstrates its effectiveness.

In summary, the main contributions of this article are:

- —We propose SpectAcle, a fault localisation framework for AI-enabled CPS that aims to identify the faulty weights of DNN controllers, by exploiting the sequences of DNN inferences. Specifically, we provide two versions of SpectAcle: SpectAcle$_{uw}$ gives equal importance to the inferences of each timestamp before the violation episode, while SpectAcle$_{w}$ weights more the inferences that are closer to the violation episode.
- —We implement both versions of SpectAcle, which are available online [37].
- —We perform experimental evaluation on 6,067 faulty versions of commonly used benchmarks of AI-enabled CPSs, controlled by DNN controllers with various structures and configurations. The results demonstrate the effectiveness of SpectAcle in identifying the faulty weights of the DNN controllers.

Article Structure. Section 2 introduces necessary preliminaries to understand the approach, and Section 3 motivates the work. Then, Section 4 introduces SpectAcle. Section 5 describes the experiment design, and Section 6 discusses experimental results. Finally, Section 7 discusses threats that may affect the validity of the approach, Section 8 reviews some related work, and Section 9 concludes the article.

2 Preliminaries

2.1 AI-Enabled CPS

Figure 1 illustrates the architecture of a typical AI-enabled CPS. The whole system $\mathcal{M}^C$ is composed of a physical plant $\mathcal{M}$ and a DNN controller C. At each timestamp t, the DNN controller C takes as input the system state $\mathbf{x}(t)$ and the external input $\mathbf{u}(t)$, and produces a control decision $\mathbf{c}(t)$ to trigger the system state evolution to the next timestamp t'. In this way, the system state $\mathbf{x}$ and the control decision $\mathbf{c}$ are both *time-variant signals*, mapping each timestamp to a vector. In particular,

some of the system variables are observable from the outside, and the evolution of these variables produces another time-variant signal $\mathbf{v}$, as the *system output*.

In this loop, the DNN controller is the primary component that decides the evolution of the system. In this article, we consider fully connected DNNs, defined as follows.

Definition 1 (DNN controller C). A DNN C includes an *input* layer, an *output* layer, and multiple *hidden* layers. At each timestamp, C takes as input a vector (jointly constituted by system state and external input) at the input layer, and computes an output scalar as the control decision at the output layer. Each hidden layer consists of a number of neurons; specifically, at hidden layer L_i ($i \in \{1, \ldots, l\}$), there are s_i neurons (where s_i is a positive integer). Each neuron has a set of parameters, including a vector of weights and a bias. We elaborate on the numerical transformations in hidden layers. Formally, the hidden layer L_i takes an s_{i-1}-dimensional vector $\vec{O}_{i-1}$ as the input of the layer, and gives an s_i-dimensional vector $\vec{O}_i$ as the output of the layer.[1] The transformation from $\vec{O}_{i-1}$ to $\vec{O}_i$ is given as follows:

$$\vec{O}_i = \sigma\left(W\vec{O}_{i-1} + \vec{b}_i\right), \tag{1}$$

where $W \in \mathbb{R}^{s_{i-1} \times s_i}$ is a matrix of *neuron weights* and $\vec{b}_i \in \mathbb{R}^{s_i}$ is a vector of *neuron bias* at layer L_i, and σ is an *activation function* that performs non-linear transformations. Common choices of activation functions include *rectified linear unit*, *Sigmoid*, *tanh*, and so on.

The output layer computes the DNN output O_c, by applying a linear transformation to $\vec{O}_l$, the output vector of the last hidden layer, which is then used as the control decision. ◁

In Equation (1), the computation from $\vec{O}_{i-1}$ to $\vec{O}_i$ consists of a linear transformation empowered by *weights* and *biases*, and a non-linear transformation based on the activation function. In this article, since the numerical transformation of each layer is primarily decided by the neuron weights, we follow the existing literature [9, 30, 47, 49] in DNN fault localisation and repair, and treat these neuron weights as the main target of our proposed fault localisation approach.

To identify a neuron weight in a DNN, we introduce the following notation. We use $W_{(i-1,j)(i,p)}$ ($i \in \{1, \ldots, l\}$, $j \in \{1, \ldots, s_{i-1}\}$ and $p \in \{1, \ldots, s_i\}$) to identify the neuron weight that has the index (p, j) in the weight matrix at the layer L_i (remember that s_i is the number of neurons at layer L_i). Intuitively, $W_{(i-1,j)(i,p)}$ is located at the pth neuron at layer L_i, used to multiply the output of the jth neuron at layer L_{i-1}, i.e., $(i-1, j)$ and (i, p), respectively, indicate the locations of the two neurons connected by the weight $W_{(i-1,j)(i,p)}$.

2.2 System Specifications

A system specification describes a property that the system must guarantee. In the domain of CPS, a widely adopted formalism is **Signal Temporal Logic (STL)** [12], that has a *formal syntax* to allow engineers to express their desired properties, and a *formal semantics* that allows to check whether the system output satisfies the specification.

Definition 2 (STL Syntax). Let $\vec{v}$ be an N-dimensional vector. *Atomic predicates* are represented as $\alpha ::\equiv (f(\vec{v}) > 0)$, where f is a function that maps $\vec{v}$ to a real number. The syntax of an STL formula φ is defined as follows:

$$\varphi :\equiv \alpha \mid \bot \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \Box_I \varphi \mid \Diamond_I \varphi \mid \varphi_1 \,\mathcal{U}_I\, \varphi_2.$$

Here, I is a closed time interval $[a, b]$, where $a, b \in \mathbb{R}$ and $a < b$. Note that we only consider specifications with bounded time, such that the specification satisfaction can always be assessed.

[1] Specifically, s_0 is the dimension of the input layer, and $\vec{O}_0$ is the input of the DNN from the input layer.

$\Box_I$, $\Diamond_I$ and $\mathcal{U}_I$ are temporal operators *always*, *eventually* and *until*, which allow to express complex temporal properties. ◁

An advantage of STL is its quantitative robust semantics [12, 17] that tells not only *whether* a system output $\mathbf{v}$ satisfies a specification φ but also *how robustly* it does so. Next, we give the definition of quantitative robust semantics of STL, that maps a system output signal $\mathbf{v}$ and an STL formula to a real number. Here, the signal $\mathbf{v}\colon [0, \mathrm{T}] \to \mathbb{R}^N$ is a time-variant function that represents an execution trace of the system, where T is the time horizon, and N is the dimension of $\mathbf{v}$.

Definition 3 (Quantitative Robust Semantics). Given a system output $\mathbf{v}$ and an STL formula φ, the *robust semantics* provides a quantity $[[\mathbf{v}, \varphi]] \in \mathbb{R} \cup \{\infty, -\infty\}$ that tells *how robustly* $\mathbf{v}$ satisfies or violates φ. The semantics is defined as follows:

$$
\begin{array}{llll}
[[\mathbf{v}, \alpha]] & := f(\mathbf{v}(0)) & [[\mathbf{v}, \bot]] & := -\infty \qquad [[\mathbf{v}, \neg\varphi]] := -[[\mathbf{v}, \varphi]] \\
[[\mathbf{v}, \varphi_1 \wedge \varphi_2]] & := \min\Big([[\mathbf{v}, \varphi_1]], [[\mathbf{v}, \varphi_2]]\Big) & [[\mathbf{v}, \varphi_1 \vee \varphi_2]] & := \max\Big([[\mathbf{v}, \varphi_1]], [[\mathbf{v}, \varphi_2]]\Big) \\
[[\mathbf{v}, \Box_I \varphi]] & := \inf_{t \in I}\Big([[\mathbf{v}^t, \varphi]]\Big) & [[\mathbf{v}, \Diamond_I \varphi]] & := \sup_{t \in I}\Big([[\mathbf{v}^t, \varphi]]\Big) \\
[[\mathbf{v}, \varphi_1 \,\mathcal{U}_I\, \varphi_2]] & := \sup_{t \in I}\Big(\min\Big([[\mathbf{v}^t, \varphi_2]], \inf_{t' \in [0,t)} [[\mathbf{v}^{t'}, \varphi_1]]\Big)\Big) &&
\end{array}
,
$$

where $\mathbf{v}^t$ denotes the t-shift of $\mathbf{v}$, namely, $\mathbf{v}^t$ maps a timestamp $t' \in [0, \mathrm{T} - t]$ to $\mathbf{v}^t(t') = \mathbf{v}(t + t')$.

By the robust semantics, we can infer the satisfaction of $\mathbf{v}$ to φ: if $[[\mathbf{v}, \varphi]]$ is positive, it means that $\mathbf{v}$ satisfies φ (i.e., $\mathbf{v} \models \varphi$); if $[[\mathbf{v}, \varphi]]$ is negative, it means that $\mathbf{v}$ violates φ (i.e., $\mathbf{v} \not\models \varphi$). ◁

3 Motivation

In this work, since the DNN controller is the primary component that decides the evolution of a system, we assume that the system executions that violate the specification should be attributed to erroneous control decisions given by the DNN controller. Consequently, we take DNN controllers as the target of our fault localisation.

For single-inference DNNs, fault localisation approaches have been successfully applied to identify the causes of wrong DNN inferences. Eniser et al. [15] proposed DeepFault, which leverages SBFL to construct execution spectra of neurons and utilises existing suspiciousness metrics to identify the suspicious neurons responsible for the misclassification of DNN. Sohn et al. [47], in their DNN repair tool Arachne, proposed a bidirectional localisation approach to identify the faulty weights, which uses both *gradient losses* and *forward impacts* of each weight to characterise the relevance of the weight to the final DNN output. However, these approaches cannot be applied in the context of AI-enabled CPS. This is due to two main issues:

— Both DeepFault and Arachne require a notion of *correctness* of a DNN inference, which relies on the ground-truth labels of DNN inputs. However, in an AI-enabled CPS, it is not possible to determine whether a single inference of the DNN controller is correct or not, i.e., there is no such an explicit ground truth. To assess the correctness of DNN inferences, we can only rely on the *effect* they have on the satisfaction of the system specification (see Section 2.2), i.e., whether they provide control decisions that guarantee the specification satisfaction of system executions. In this sense, the satisfaction of the system specification provides an *indirect* ground truth regarding the correctness of the inference of the DNN controller.

— Moreover, while DeepFault and Arachne only consider individual DNN inferences, the DNN controller of an AI-enabled CPS performs a 'sequence of inferences' (as shown by Figure 1) that drive the evolution of the plant. These inferences must be considered together in fault localisation.

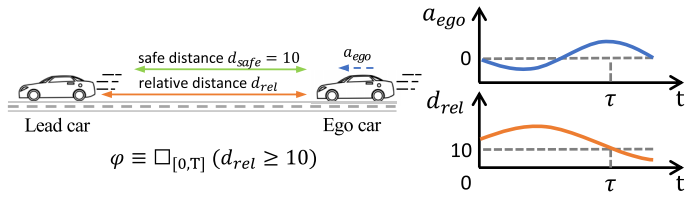

Fig. 2. Example—Adaptive cruise control system that violates the specification $\varphi \equiv \square_{[0,T]}(d_{rel} \geq 10)$.

Example 1. Consider the AI-enabled **Adaptive Cruise Control (ACC)** system introduced in Section 5.2 where the DNN controller C produces the acceleration a_{ego} that determines the motion of the ego car and therefore affects the relative distance d_{rel} between the ego car and the lead car. The system specification φ requires that d_{rel} should always be greater than the safe distance $d_{safe} = 10$, during the time interval $[0, T]$; formally, $\varphi \equiv \square_{[0,T]}(d_{rel} \geq 10)$. While we cannot tell whether a given acceleration command is correct or not, we can check how it affects the relative distance d_{rel}: if this violates the system specification, the acceleration command can be deemed not correct. Figure 2 shows examples of execution traces of a_{ego} and d_{rel}. τ is the timestamp that identifies the beginning of the violation; therefore, the control decisions that lead to the violation must precede τ. Given the inference process of DNN (as introduced in Definition 1), these erroneous control decisions originate from a number of sub-optimal DNN parameters, and therefore, our primary focus in fault localisation is to identify these parameters that account for the specification violation. While the exact timestamps at which the controller makes incorrect decisions are not known, it is feasible to assess the impacts of each parameter to a control decision and thereby select the parameters that exert the greatest impact on the control decisions. These parameters can be deemed to be more relevant to the erroneous control decisions, hence they can be further considered as more responsible for the specification violation of system executions. ◁

Problem Statement. In this article, we consider an AI-enabled CPS that violates a specification, and the specification violation is due to erroneous control decisions made by the DNN controller. These erroneous decisions result from sub-optimal settings of some DNN *parameters* that significantly influence the DNN inferences, leading to the generation of erroneous decisions. Specifically, we consider *neuron weights* (see Definition 1) as our target parameters, following existing literature [47] on DNN classifiers, as these neuron weights can be directly modified by using DNN repair approaches [9, 36, 47] that take as input fault localisation results for DNN repair.

4 SpectAcle

We assume to have an AI-enabled CPS $\mathcal{M}^C$ that violates a system specification φ; the violation has been determined by a test suite $\mathcal{S}$ of input signals $\mathbf{u}$, each of which is executed for an execution time T (in practice, the time horizon T of signals can be decided based on the specification φ, such that the satisfaction of φ can be assessed (see Definitions 2 and 3)).

We propose that fault localisation approach SpectAcle, whose aim is to detect the *cause* of the specification violation in the DNN controllers. Figure 3 provides an overview of SpectAcle. The approach consists of four phases:

(1) *Test Suite Execution*: It monitors the system executions of the tests in $\mathcal{S}$ and obtains the sequences of DNN controller inferences before the specification violation for the failing tests (as test n in Figure 3), or the whole execution if no violation occurs (as test 1 in Figure 3).

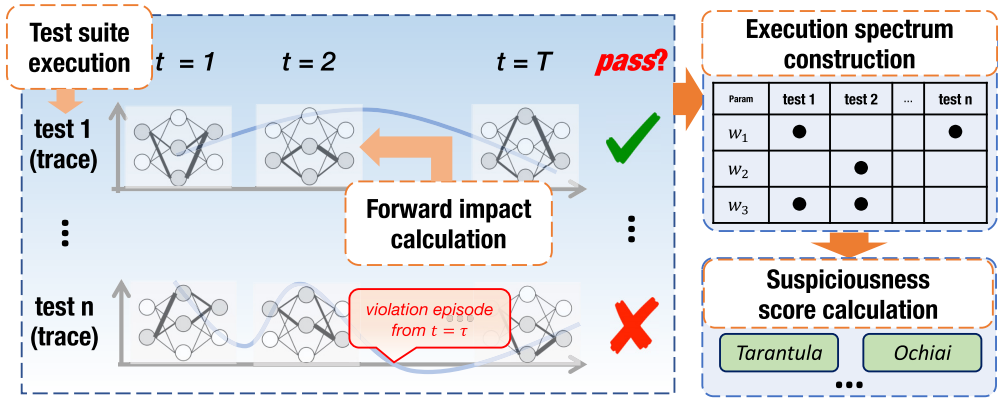

Fig. 3. SpectAcle—Workflow of the approach.

(2) *Forward Impact Calculation*: Based on the system executions for the tests, it calculates the *forward impact* of each DNN weight, which indicates the relevance of a DNN weight to the final DNN output (i.e., a control decision).
(3) *Execution Spectrum Construction*: By using the values of the forward impact in passing and failing tests, it builds a *spectrum* for each weight w, which tells how often w was *relevant* in passing and failing tests.
(4) *Suspiciousness Score Calculation*: Starting from the spectra of the different weights, it uses suspiciousness metrics borrowed from *SBFL* to identify the weights that are more likely responsible for the specification violations.

Algorithm 1 presents the four phases of SpectAcle. The algorithm takes as input an AI-enabled CPS $\mathcal{M}^C$, a system specification φ, an execution time T, a test suite $\mathcal{S}$ of input signals $\mathbf{u}$ for $\mathcal{M}^C$, the number of top k weights to consider in each hidden layer, the number s of suspicious weights to return in the final results and a suspiciousness metric `SusMetr`.

The following sections describe in detail the four phases of SpectAcle.

4.1 Test Suite Execution

SpectAcle first initialises an empty execution spectrum ES_w with four attributes $\langle ep, np, ef, nf \rangle$ for each weight w of the DNN controller C (Lines 1–3). The meaning of these attributes and how these attributes are updated will be described in *Execution spectrum construction* in Section 4.3.

Then, it takes each test case $\mathbf{u}$ in the test suite $\mathcal{S}$ (Line 4), and feeds it to the system $\mathcal{M}^C$ for system execution (Line 5). The execution returns the system output $\mathbf{v}$ and the sequence InfSeq of inferences of the DNN controller C. Each infer_t in InfSeq contains the information regarding a single DNN inference at time t: the DNN input, the output $O_{i,j}$ of each neuron $n_{i,j}$ and the final output O_c of the DNN that is used as the control action $\mathbf{c}(t)$ for the plant $\mathcal{M}$. The neuron outputs and the final DNN output are used for the forward impact calculations (see Section 4.2).

Given the output signal $\mathbf{v}$, it assesses whether the specification φ is satisfied or violated (Line 6). Moreover, in case of violation, it analyses the execution trace and obtains the timestamp τ when the system output $\mathbf{v}$ starts to violate the specification; the DNN controller must have done some erroneous decisions before the timestamp τ, i.e., in $[0, \tau]$. For example, in Example 1, since d_{rel} starts to be below the safe distance 10 at τ, τ is then the first timestamp when we start to observe the violation of the specification. If the system execution satisfies the specification, τ is set as the

Algorithm 1: SpectAcle – The proposed approach

Input : an AI-enabled CPS $\mathcal{M}^C$ with a DNN controller C;
a set $\mathcal{W}$ of neuron weights (the subset of which at the hidden layers L_i denoted as W_i);
a system specification φ;
maximum execution time T;
a test suite $\mathcal{S}$;
number k of top weights to consider in each hidden layer;
number s of weights to return as suspicious;
a suspiciousness metric `SusMetr`

Output: a set of suspicious weights Θ_s

1 **foreach** $W_i \in \{W_1, \dots W_l\}$ **do**
2 **foreach** $w \in W_i$ **do**
3 $ES_w \leftarrow \langle 0, 0, 0, 0 \rangle$ // initial spectrum of each weight
4 **foreach** $\mathbf{u} \in \mathcal{S}$ **do**
5 $\langle \mathbf{v}, \mathsf{InfSeq} \rangle \leftarrow \mathcal{M}^C(\mathbf{u})$ // system execution
6 identify τ as $\begin{cases} \text{the start of violation episode, if } \mathbf{v} \not\models \varphi \\ \text{the execution time T, otherwise} \end{cases}$
7 $\beta \leftarrow \frac{\mathrm{T}}{\int_0^\tau \Omega(t)\,dt}$ // compute scaling factor
8 **foreach** $t \in [0, \tau]$ **do**
9 $\mathsf{infer}_t \leftarrow \mathsf{InfSeq}(t)$ // obtain the details of the inference at each timestamp
10 **foreach** $W_i \in \{W_1, \dots W_l\}$ **do**
11 **foreach** $w \in W_i$ **do**
12 $I_w \leftarrow \mathtt{CalculateForwardImpact}(w, \mathsf{infer}_t)$
13 $top_k^i \leftarrow \text{sort } \{I_w \mid w \in W_i\}$ // calculate forward impacts and select the top k weights in each hidden layer by their forward impacts
14 **foreach** $w \in W_i$ **do**
15 $\begin{cases} ES_w(ep) \leftarrow ES_w(ep) + 1 & \mathbf{v} \models \varphi \wedge w \in top_k^i \\ ES_w(np) \leftarrow ES_w(np) + 1 & \mathbf{v} \models \varphi \wedge w \notin top_k^i \\ ES_w(ef) \leftarrow ES_w(ef) + \beta \cdot \Omega(t) & \mathbf{v} \not\models \varphi \wedge w \in top_k^i \\ ES_w(nf) \leftarrow ES_w(nf) + \beta \cdot \Omega(t) & \mathbf{v} \not\models \varphi \wedge w \notin top_k^i \end{cases}$ // construct execution spectrum
16 **foreach** $w \in \mathcal{W}$ **do**
17 $score_w \leftarrow \mathtt{CalculateSuspiciousness}_{\mathtt{SusMetr}}(ES_w)$
18 $\Theta_s \leftarrow \text{sort } \{score_w \mid w \in \mathcal{W}\}$ // select the top s suspicious weights in a descending order

simulation time T. In order to identify τ, we use the technique of *trace diagnostics* proposed by Bartocci et al. [5].

4.2 Forward Impact Calculation

In the interval $[0, \tau]$, the DNN controller performs a sequence of inferences to determine the control decision at each timestamp, which leads to the satisfaction or violation of the specification. SpectAcle, for each timestamp $t \in [0, \tau]$ (Line 8), computes the *forward impact* of each DNN weight in this interval to determine its contribution to the DNN output, namely, the control decision at t (Lines 9–12). The calculation follows the approach proposed by Sohn et al. [47]; see an illustration of the calculation in Figure 4. In order for a weight $W_{(i-1,j-1)(i,j)}$ to have an impact on the control decision of a single inference infer_t, the following conditions must occur: (1) the weight must affect the linear output of its associated neuron $n_{i,j}$; (2) the neuron output $O_{i,j}$ of its associated neuron

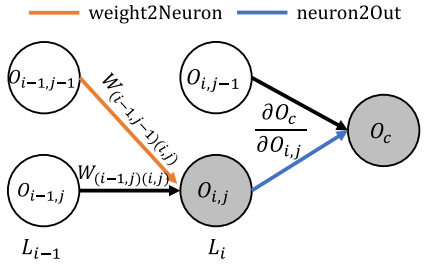

Fig. 4. Forward impact of $W_{(i-1,j-1)(i,j)}$ to the final DNN output O_c.

$n_{i,j}$ should contribute to the final DNN output of the current inference. Hence, the forward impact of a weight on an inference is given by two key components:

— *weight2Neuron* of a weight $W_{(i-1,j-1)(i,j)}$ is the impact on the output of its neuron $n_{i,j}$, formally:

$$weight2Neuron = \frac{W_{(i-1,j-1)(i,j)} \times O_{i-1,j-1}}{\sum_{m=1}^{s_{i-1}} \left| W_{(i-1,m)(i,j)} \times O_{i-1,m} \right|}$$

i.e., the ratio of the product of the weight and the output value of the neuron in the previous layer it is connected to, over the sum of the absolute value of the element-wise product between the output values of the neurons in the previous layer and the corresponding weights of the current neuron. Intuitively, it determines the relative contribution of weight $W_{(i-1,j-1)(i,j)}$ to the neuron output $O_{i,j}$ of neuron $n_{i,j}$.

— *neuron2Out* is the impact of the output $O_{i,j}$ of neuron $n_{i,j}$ on the final DNN output O_c, i.e., the control decision of the DNN controller, formally:

$$neuron2Out = \frac{\partial O_c}{\partial O_{i,j}}$$

i.e., the gradient of O_c with respect to the neuron output $O_{i,j}$.

The forward impact $I_{W_{(i-1,j-1)(i,j)}}$ of a weight $W_{(i-1,j-1)(i,j)}$ on the control decision can be represented as the absolute value of the multiplication of the two elements, namely:

$$I_{W_{(i-1,j-1)(i,j)}} = \left| weight2Neuron \times neuron2Out \right|.$$

4.3 Execution Spectrum Construction

After the computation of the forward impact, SpectAcle constructs an *execution spectrum* $ES_w = \langle ep, np, ef, nf \rangle$ for each weight w, that represents the contributions of w to the satisfaction and violation of system executions. The execution spectrum considers both the specification satisfaction/violation, and the contributions of w to the system execution identified by its forward impact (see Section 4.2); these elements are reflected in the naming of the spectrum components:

— p (for *p*ass) and f (for *f*ail) denote whether a system execution satisfies the system specification or not;

— e (for *e*xecuted) and n (for *n*ot executed) denote whether the weight is *executed* or not, i.e., if it contributes to the satisfaction/violation of the system execution. It decides this as follows: it sorts the weights of each hidden layer L_i in a descending order according to their forward impact (Line 13); if a weight w is ranked in the top k greatest weights (labelled as top_k^i) in layer L_i, then w will be deemed as *executed*.

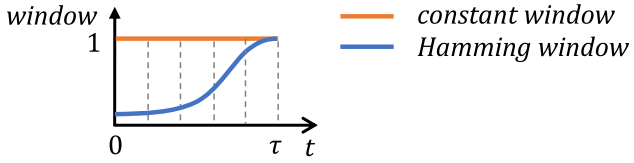

Fig. 5. Window functions.

For example, attribute *ep* records the number of times a weight is ranked in the top k (thus *executed*) and its corresponding inference is situated in a 'safe' system execution (thus *pass*).

So, in Algorithm 1, for each weight w, the corresponding attribute in $\langle ep, np, ef, nf \rangle$ is incremented (Line 15) after one system execution:

- if $\mathbf{v} \models \varphi \wedge w \in top_k^i$, i.e., the specification is satisfied and w is ranked in the top k, ep is incremented;
- if $\mathbf{v} \models \varphi \wedge w \notin top_k^i$, i.e., the specification is satisfied and w is not ranked in the top k, np is incremented;
- if $\mathbf{v} \not\models \varphi \wedge w \in top_k^i$, i.e., the specification is violated and w is ranked in the top k, ef is incremented;
- if $\mathbf{v} \not\models \varphi \wedge w \notin top_k^i$, i.e., the specification is violated and w is not ranked in the top k, nf is incremented.

In the update of the spectrum components (Line 15), a spectrum component is incremented by a quantity $\Omega(t)$ called *window*, that depends on the time when the inference occurs, and it is scaled by a factor β that depends on the length of the test (namely, the time horizon T of the signal). In the following, Section 4.3.1 provides two versions of Ω, and Section 4.3.2 explains the scaling factor computation.

4.3.1 Two Approaches for Spectrum Construction. Recall from Section 4.1 that the timestamps when the DNN controller makes erroneous decisions are before the timestamp τ that identifies the beginning of the specification violation, and, therefore, fault localisation should focus on the weights that are relevant in the time interval $[0, \tau]$. However, it is difficult to exactly determine *when* the wrong decisions are made, and so decide which timestamps should be considered in fault localisation. To this end, we propose two versions of SpectAcle (SpectAcle$_{\text{UW}}$ and SpectAcle$_{\text{W}}$) that make different assumptions on which timestamps should be considered. This is reflected in the definition of a window function Ω used in spectrum construction (see Line 15), that increments the spectrum depending on the specific timestamp:

$$\Omega(t) = \begin{cases} 1 & \text{in SpectAcle}_{\text{UW}} \\ \Omega_{\mathsf{ham}}(t) & \text{in SpectAcle}_{\text{W}} \end{cases}$$

SpectAcle$_{\text{UW}}$ considers in the same way all the relevant weights of all the timestamps (it uses a *constant window*), while SpectAcle$_{\text{W}}$ weights more the neuron weights that were relevant closer to the system violation (it uses a *Hamming window*), as explained in the following. These two types of windows are visualised in Figure 5.

SpectAcle$_{UW}$ (Unweighted SpectAcle). SpectAcle$_{\text{UW}}$ considers every infer_t in the DNN inference sequence InfSeq as having equal relevance to the violation or satisfaction. So, the value added to a spectrum component is always 1, regardless of the distance of the timestamp t of the inference to the beginning of specification violation τ. In this sense, SpectAcle is *unweighted*.

Table 1. Suspiciousness Metrics `SusMetr` Used in This Article

Tarantula [26]	Ochiai [2]	D* [55]	Jaccard [1]	Kulczynski2 [45]
$\frac{\frac{ef}{ef+nf}}{\frac{ef}{ef+nf}+\frac{ep}{ep+np}}$	$\frac{ef}{\sqrt{(ef+nf)(ef+ep)}}$	$\frac{ef^*}{ep+nf}$	$\frac{ef}{ef+nf+ep}$	$\frac{1}{2}\left(\frac{ef}{ef+nf}+\frac{ef}{ef+ep}\right)$

$* > 0$. Here we set $*$ as 3, according to previous practices [55, 56].

SpectAcle$_W$ (Weighted SpectAcle). Differently from the unweighted SpectAcle, SpectAcle$_\text{W}$ emphasises the weights that are relevant at the timestamps that are closer to the beginning of the specification violation τ. In this sense, SpectAcle is *weighted*. Hence, SpectAcle$_\text{W}$ requires a time-related window function; to that end, we select the *Hamming window*, which is widely applied in signal processing and spectral analysis to smooth data and reduce noise [3]. The Hamming window Ω_{ham} is defined as

$$\Omega_{\mathsf{ham}}(t) = 0.54 - 0.46 \cdot \cos\left(\frac{\pi \cdot t}{\tau}\right) \quad \text{with } t \in [0, \tau].$$

When $t = \tau$, the Hamming window has the maximum value 1; instead, when $t = 0$, it has the minimum value 0.08.

4.3.2 Scaling Factor. Note that different tests $\mathbf{u}$ contribute with different numbers of inferences to the construction of the spectrum. For passing tests, all the inferences till T are considered, while for failing tests, all the inferences till τ ($\tau \leq \text{T}$) are considered. To ensure that each test contributes equally to the construction of the spectrum, the window value in failing tests is *scaled* depending on the length of the test. Specifically, when increasing the spectrum of *ef* and *nf* at Line 15 of Algorithm 1, this is scaled by the *scaling factor* β computed at Line 7. Intuitively, the spectrum additions for inferences of the shorter tests (i.e., the failing ones) are weighted more than those of longer tests (i.e., the passing ones).

The computation of the scaling factor depends on the window function used. Let us consider the constant window used in SpectAcle$_\text{UW}$. In this case, the scaling factor becomes $\frac{\text{T}}{\int_0^\tau 1\,dt} = \frac{\text{T}}{\tau}$. For example, the scale factor of a test failing at $\tau = \text{T}/2$ is $\beta = \text{T}/(\text{T}/2) = 2$; in this way, a failing test with half of the length of a passing test provides the same contribution to the spectrum. In the case of the Hamming window, the scaling factor can be generalised to $\frac{\text{T}}{\int_0^\tau \Omega_{\mathsf{ham}}(t)\,dt}$.

4.4 Suspiciousness Score Calculation

Based on the execution spectra of all weights, SpectAcle uses *suspiciousness metrics* to calculate the *suspiciousness score* of each weight, to identify the weights that are highly relevant to the violations. Table 1 reports five suspiciousness metrics that have been widely used in software fault localisation [57].[2] The basic premise behind these metrics is that the more frequently a weight is ranked in top k when the violation occurs, and the less frequently it is ranked in top k when the satisfaction happens, the more suspicious the weight is. A higher suspiciousness score means that the weight is more likely to be a faulty one. Lines 16–18 describe the process of *suspiciousness score calculation*. Based on the execution spectrum ES_w obtained by Algorithm 1, SpectAcle calculates the suspiciousness score for each w based on a given suspiciousness metric `SusMetr` (Lines 16 and 17) and selects the top s weights that have the highest suspiciousness scores as the output of

[2]Note that we did not use the metric Kulczynski1 [45], as it has been shown that it is equivalent to Jaccard (see Proposition 7.2 in [45]).

Table 2. An Illustrative Example of SpectAcle$_{UW}$ (Grey Cells Indicate Top-k)

		Forward Impact				Exec. Spectrum from an Exec.	Exec. Spectrum $\langle ep, np, ef, nf \rangle$	Suspiciousness Score
		t_0	t_1	t_2	t_3			
w_1	$\mathbf{u}_1$	**0.7**	**0.9**	0.6	**1.1**	$\langle 3, 1, 0, 0 \rangle$	$\langle 3, 1, 0, 4 \rangle$	0
	$\mathbf{u}_2$	0	0.1	0.3	×	$\frac{4}{3} \cdot \langle 0, 0, 0, 3 \rangle$		
w_2	$\mathbf{u}_1$	0.3	0.5	**0.8**	0.2	$\langle 1, 3, 0, 0 \rangle$	$\langle 1, 3, 4, 0 \rangle$	0.8
	$\mathbf{u}_2$	**0.4**	**0.9**	**2.3**	×	$\frac{4}{3} \cdot \langle 0, 0, 3, 0 \rangle$		

fault localisation (Line 18), because these are the weights that are more likely responsible for the violations.

Example 2. For explanation purposes, we show how SpectAcle$_{UW}$ (that uses the constant window) works on a toy example; the application of SpectAcle$_{W}$ would be similar. We consider a DNN controller having only one hidden layer that includes two neurons, each having only one weight. Table 2 shows the computation of fault localisation for it. The system executes in time interval $[t_0, t_3]$. The test suite $\mathcal{S}$ consists of two input signals $\mathbf{u}_1$ and $\mathbf{u}_2$. $\mathbf{u}_1$ triggers the inference sequence InfSeq_1 and gives the system output $\mathbf{v}_1$ that satisfies specification φ; $\mathbf{u}_2$ triggers the inference sequence InfSeq_2 and gives the system output $\mathbf{v}_2$ that violates φ at t_2. As shown in Line 6 of Algorithm 1, we need to identify the timestamp τ that is either when the system output starts to violate the specification, or the simulation time T; as a result, we identify τ_1 and τ_2, respectively, for $\mathbf{u}_1$ and $\mathbf{u}_2$, namely, they are $\tau_1 = t_3$ and $\tau_2 = t_2$. Table 2 also reports the *forward impact* of weights w_1 and w_2 for each DNN inference at each timestamp.

We set the hyper-parameter k in Line 13 as 1, i.e., the weight whose *forward impact* is ranked in top-1 at the hidden layer will be regarded as *executed* (in grey in the table). For instance, for InfSeq_1, since the *forward impact* of w_1 at t_1 is 0.9, higher than 0.5 of w_2, w_1 is in the top-1 at t_1; hence, *ep* of w_1 and *np* of w_2 are incremented in the spectrum. Considering all the four timestamps, the execution spectra of w_1 and w_2 for InfSeq_1 are $\langle 3, 1, 0, 0 \rangle$ and $\langle 1, 3, 0, 0 \rangle$.

In this example, SpectAcle$_{UW}$ uses a constant window (see Section 4.3.1) to assign equivalent significance to DNN inferences at different timestamps. For the sequence InfSeq_2 (that leads to the violation of φ at $\tau_2 = t_2$), the constant window gives the same contribution of 1 at t_0, t_1, and t_2. Since the *forward impact* of w_2 is consistently greater than that of w_1, we consider w_2 as always executed; so, the execution spectrum of w_2 for InfSeq_2 is $\langle 0, 0, 3, 0 \rangle$. The scaling factor β is used to keep the same contribution to the spectrum between passing and failing sequences. For InfSeq_2 that leads to specification violation at $\tau_2 = t_2$, β is $\frac{|\{t_0,\ldots,t_3\}|}{|\{t_0,\ldots,t_2\}|} = \frac{4}{3}$.

In this example, we use Jaccard as suspiciousness metric and we set $s = 1$. The score of w_1 is 0 and the score of w_2 is 0.8, so SpectAcle$_{UW}$ returns w_2 as the suspicious weight. ◁

5 Experiment Setup

In this section, we elaborate on the detailed experimental setup to investigate the effectiveness of SpectAcle. All AI-enabled CPSs used as benchmarks and the source code of SpectAcle for reproducing our experiments are publicly available on the GitHub repository [37]. Additional plots not reported in the article are available at [38].

5.1 Research Questions (RQ)

We identified six RQs to assess the effectiveness of SpectAcle.

Table 3. Benchmark AI-Enabled CPSs—CPSs and Their DNN Controllers ($|\mathcal{W}|$ Is the Number of Weights of C; $|\mathcal{M}|$ Is the Number of Blocks of $\mathcal{M}$)

CPS	Description	C's Inputs		C's Output	$\mathcal{M}^C$	C's Structure	$\lvert\mathcal{W}\rvert$	$\lvert\mathcal{M}\rvert$
		u (Range [Lower, Upper])	**x**					
ACC	Keep relative distance and ego car speed	Lead car's acceleration [-1, 1]	Ego car's speed	Ego car's acceleration	ACC#1 ACC#2	[10, 10, 10] [15, 15, 15]	200 450	49 49
AFC	Maintain an optimal air-to-fuel ratio	Throttle angle [8.8, 61.1]; engine speed [900, 1100]	Inlet air mass flow rate; air-to-fuel ratio	Commanded fuel	AFC#1 AFC#2	[15, 15, 15] [15, 15, 15, 15]	450 675	153 153
SC	Stabilize the pressure	Steam flowrate [3.99, 4.01]	Actual pressure	Cooling water flow rate	SC#1 SC#2 SC_R	[10, 10, 10, 10] [15, 15, 15, 15] [3]	300 675 18	55 55 55
WT	Monitor the water level	Water level reference [10, 20]	Actual water level	Waterflow	WT#1 WT#2	[5, 5, 5] [15, 15, 15]	50 450	11 11

AFC, Abstract Fuel Control; SC, Steam Condenser; WT, Water Tank.

RQ1: *Does SpectAcle effectively localise the faulty DNN weights in AI-enabled CPSs?*
In this RQ, we evaluate whether SpectAcle can actually identify faulty weights that exist in a DNN controller C. To do this, we compare it with two baseline approaches.

RQ2: *How does the selection of hyper-parameter k influence the effectiveness of SpectAcle?*
In this RQ, we investigate which value of k (used to consider a weight 'executed' or not in spectrum construction. See Section 4.3 and Line 13 in Algorithm 1) allows to obtain the best fault localisation results.

RQ3: *How do SpectAcle$_W$ and SpectAcle$_{UW}$ compare with each other?*
In this RQ, we investigate the influence of the window function (see Section 4.3.1) to the effectiveness of SpectAcle. Namely, we compare the performances of SpectAcle$_{\text{W}}$ with that of SpectAcle$_{\text{UW}}$.

RQ4: *How does the selection of the suspiciousness metric* `SusMetr` *influence the effectiveness of SpectAcle?*
In this RQ, we investigate which suspiciousness metric, among the five reported in Table 1, allows to obtain the best fault localisation results.

RQ5: *Are the weights identified by SpectAcle useful for improving the DNN controller performance?*
In this RQ, we investigate whether the performance of the DNN controller can be improved by repairing the weights identified by SpectAcle.

RQ6: *What is the computational cost of SpectAcle?*
In this RQ, we investigate if the computational cost of executing SpectAcle is acceptable and if it differs across different AI-enabled CPSs.

5.2 Benchmarks

To evaluate the effectiveness of SpectAcle, we selected nine AI-enabled CPSs. Table 3 reports their functionalities, controller details, and plant details.

The AI-enabled CPSs have been built as follows. We first selected, as plants $\mathcal{M}$, four CPSs largely used in the literature [23, 41, 43, 48, 51, 58, 60] that span over different domains, such as automotive and chemical industry: ACC, **Abstract Fuel Control (AFC)**, **Steam Condenser (SC)**, and **Water Tank (WT)**. All the subject CPSs are implemented in Simulink [42], the de facto standard to design and model CPSs. All these CPSs are embedded with a classic controller, that we want to replace with a DNN controller. In the literature, for each of these CPSs, some STL specifications describing their expected behaviour have been reported. We selected some specifications for each CPS and

tuned their parameters (e.g., constants used as thresholds), so that the classic controller always behaves correctly over the input signals $\mathbf{u}$ that we use to train the DNN controller (as the classic controller must act as a supervisor during training. See Section 5.2.1).

ACC. ACC [41] aims to maintain a safe distance between the ego car and the lead car by adjusting the acceleration of the ego car. The ACC system takes the lead car's acceleration as input and provides outputs in terms of the velocities and distances covered by both the lead and ego cars. The specification φ^1_{ACC} requires that the relative distance d_{rel} between two cars must always be higher than the sum of the safe distance d_{safe} and the *reaction distance reactDist* (i.e., $1.4 \times v_{ego}$, where 1.4 is the *time-gap* [31]) of the ego car during the time interval $[0, 50]$, and that the speed of the ego car v_{ego} should always be lower than a given speed v_{set}. Here, d_{safe} and v_{set} are set to 10 and 30, respectively.

$$\varphi^1_{\text{ACC}} \equiv \square_{[0,50]}((d_{rel} \geq d_{safe} + reactDist) \wedge v_{ego} \leq v_{set}) \quad \text{with } reactDist = 1.4 \times v_{ego}.$$

The second specification φ^2_{ACC} states that during the time interval $[0, 45]$, if the relative distance d_{rel} is lower than the sum of the reaction distance *reactDist* of the ego car and a distance slightly above the safe distance d_{safe} (we use 12 in the specification), the system is on the edge of a hazardous state. The system is required to recover from this state and return to a safe state within 5 seconds.

$$\varphi^2_{\text{ACC}} \equiv \square_{[0,45]}(d_{rel} < 12 + reactDist \rightarrow \diamond_{[0,5]}(d_{rel} \geq 12 + reactDist)) \quad \text{with } reactDist = 1.4 \times v_{ego}.$$

AFC. AFC, developed by Toyota [23], aims to regulate the mixture ratio of air and fuel in the engine intake for optimal combustion efficiency. The system takes the pedal angle and engine speed as exogenous inputs and minimises the deviation $\mu = |AF - AF_{ref}|/AF_{ref}$, which is expressed as the absolute difference between the actual air fuel ratio AF and its reference value AF_{ref} divided by AF_{ref}. With the exogenous signals, the controller dynamically adjusts intake of air and fuel, to maintain the optimal air-fuel ratio. The specification φ^1_{AFC} requires that the deviation μ should never exceed a predefined value μ_{set} during the time interval $[0, 30]$. Here, μ_{set} is set to 0.2.

$$\varphi^1_{\text{AFC}} \equiv \square_{[0,30]}(\mu \leq \mu_{set}).$$

The second specification φ^2_{AFC} requires that, during time interval $[10, 28.5]$, when μ exceeds 0.1, the controller should bring μ back to 0.1 within 1.5 seconds.

$$\varphi^2_{\text{AFC}} \equiv \square_{[10,28.5]}(\mu > 0.1 \rightarrow \diamond_{[0,1.5]}(\mu \leq 0.1)).$$

SC. SC [58] is a component of an electric power system. During the condensation process of steam, it aims to keep the pressure in the condenser around a reference pressure. The external input of the system is the steam mass flow rate, while the system's output is the internal pressure of the condenser. The specification φ_{SC} of SC requires that the pressure should be maintained between 87 and 87.5 in the time interval $[30, 35]$.

$$\varphi_{\text{SC}} \equiv \square_{[30,35]}(pressure \geq 87 \wedge pressure \leq 87.5).$$

WT. WT [48] is used to control the inflow and outflow of water. In this system, water can enter or flow out the tank until the water level reaches a predefined reference value. Over time, the water level should stabilise and match the reference level. The system takes the reference water level h_{ref} as input and continuously outputs the actual water level h_{out}. The specification φ_{WT} requires that the absolute deviation error between h_{out} and h_{ref} remains below a predefined threshold value err_{set} during specific time intervals, namely $[4, 5]$, $[9, 10]$, and $[14, 15]$. In our experiments, we set err_{set} to 0.86.

$$\varphi_{\text{WT}} \equiv \square_{[4,5]\cup[9,10]\cup[14,15]}(|h_{out} - h_{ref}| \leq err_{set}).$$

Table 4. Training Details of the DNN Controllers C

DNN Controller C	Training Algorithm	Max Epoch	Min Gradient	Final Training Loss (MSE)
ACC#1	LMBP*	1,000	1×10^{-10}	3.18×10^{-3}
ACC#2	SCG**	1,000	1×10^{-10}	7.49×10^{-3}
AFC#1	LMBP	1,000	1×10^{-10}	1.27×10^{-5}
AFC#2	LMBP	1,000	1×10^{-10}	1.79×10^{-1}
SC#1	LMBP	1,000	1×10^{-10}	3.62×10^{-3}
SC#2	LMBP	1,000	1×10^{-10}	3.78×10^{-3}
SC_R	–	–	–	–
WT#1	BFG***	1,000	1×10^{-10}	1.84×10^{0}
WT#2	BFG	1,000	1×10^{-10}	1.83×10^{0}

*_LMBP: Levenberg–Marquardt Backpropagation (LMBP) Algorithm_ [35]; **_SCG: Scaled Conjugate Gradient backpropagation (SCG) Algorithm_ [44]; ***_BFG: BFGS quasi-Newton (BFG) Algorithm_ [19].

5.2.1 Construction of an AI-Enabled CPS. To build AI-enabled CPSs to use as a target of SpectA-cle, we proceeded as follows. In order to obtain AI-enabled CPSs $\mathcal{M}^C$, for each plant $\mathcal{M}$ described above, we considered two feed-forward DNNs as controller C; moreover, for SC, we also considered an RNN as controller C.

Some of these controllers are taken from existing repositories. Specifically, for AFC, we took the feed-forward DNNs trained in FalsifAI [60]. For SC, instead, we took the RNN controller proposed in [58]. The other controllers, instead, have been trained for this work. Specifically, for ACC, SC and WT, we followed the approach explained in [60] and trained two feed-forward neural networks for each of them. The specific training steps are detailed as follows:

(1) we randomly generate a set of 10,000 input signals $\mathbf{u}$; these input signals $\mathbf{u}$ are constrained within specific input ranges (reported in Table 3) obtained by following official documentations and technical manuals [41, 48, 58];
(2) we execute the model embedded with the classic controller over the generated input signals $\mathbf{u}$ and collect the controller output $\mathbf{c}$ and system state $\mathbf{x}$; note that, as explained before, the classic controllers never violate the system specification;
(3) the inputs $\mathbf{u}$, and the observed outputs $\mathbf{c}$ and $\mathbf{x}$ constitute the training data for the DNN controller C;
(4) we employ the *Deep Learning Toolbox* of MATLAB to train a neural network C with the aim to minimise the prediction error of $\mathbf{c}$ given input $\langle \mathbf{x}, \mathbf{u} \rangle$ for C.[3] After training and validating the neural network, we replace the classic controller in the Simulink model with the newly trained neural network controller C, so obtaining an AI-enabled CPS $\mathcal{M}^C$.

Table 4 reports details about the training of the DNN controllers (either performed by us or in the original papers) in terms of used training algorithm, maximum number of training epochs, the threshold of gradient descent for which the training stops and the final loss in terms of **Mean Squared Error (MSE)**.[4] Table 3 reports the complexities of the nine AI-enabled CPSs in terms

[3]We decided to train the neural networks in MATLAB to ease the integration with the CPS plants that are modelled in Simulink.
[4]Note that, for SC_R, we only have access to the RNN controller reported in [41, 58], but we do not have information on how it was trained and what was the final training loss.

of number of blocks of each $\mathcal{M}$ and number of weights of the controller C. From Table 4, we observe that, for most of the benchmarks, the final training loss is similar between the two DNN controllers; the only difference is AFC, for which the most complex controller (i.e., AFC#2) obtains a larger loss; this is probably due to the fact that AFC#2 is a much more complex network than AFC#1.

In the following, a *correct benchmark* is given by an AI-enabled CPS $\mathcal{M}^C$ and a specification φ, identified as $\mathcal{M}^C\_\varphi$. Therefore, in total, there are 13 correct benchmarks (there are four correct benchmarks for both ACC and AFC, three correct benchmarks for SC, and two correct benchmarks for WT).

5.2.2 Faulty Benchmarks. To evaluate the effectiveness of SpectAcle, we introduce artificial faults in C and we check the ability of SpectAcle and of the compared baseline approaches in identifying them. To do this, we mutate neuron weights to produce faulty DNN controllers, similarly to what was done by Sohn et al. [47] to mimic improper training of the weights. In order to measure 'how much' a faulty controller affects the correctness of the system, we define the *safety rate* that counts how many tests of the test suite satisfy the specification. Formally, given a test suite $\mathcal{S}$, the safety rate SR of an AI-enabled CPS $\mathcal{M}^C$ w.r.t. a specification φ is defined as

$$SR(\mathcal{M}^C, \varphi, \mathcal{S}) = \frac{|\{\mathbf{u} \in \mathcal{S} \mid \mathcal{M}^C(\mathbf{u}) \models \varphi\}|}{|\mathcal{S}|} \tag{2}$$

i.e., the ratio of tests satisfying the specification.

Given a correct benchmark $\mathcal{M}^C\_\varphi$ (see Section 5.2.1), the generation of *faulty benchmarks* containing different number of faults is as follows.

First, we generate a test suite $\mathcal{S}$ of 100 tests (i.e., input signals $\mathbf{u}$) for the original AI-enabled CPS. Since the time domain is continuous, it is not possible to synthesise an input signal by providing its value at each moment; therefore, we adopt a commonly used parametrised representation, namely *piecewise constant*, in which the time domain is split in H intervals, and a signal is given by H constants, each identifying the value of the signal in a time interval. We generate the tests by uniformly sampling the values for the H intervals. We set H by following related literature [48, 60]: following [60], H has been set to 10 for ACC, and to 5 for AFC; following [48], H has been set to 7 for SC, and to 3 for WT. $\mathcal{M}^C$ passes all the tests, i.e., the safety rate $SR(\mathcal{M}^C, \varphi, \mathcal{S})$ is 100%.

Then, we produce a set of *faulty benchmarks* containing single faults as follows:

- We identify the range of weights values in the original controller C, i.e., $[W_{min}, W_{max}]$;
- For each weight w of C:
  - we sample a value v' in the original range of weights enlarged of 50%, i.e., $v' \in [W_{min} - \frac{\mathsf{sgn}(W_{min}) \times W_{min}}{2}, W_{max} + \frac{\mathsf{sgn}(W_{max}) \times W_{max}}{2}]$, where $\mathsf{sgn}(\cdot) \in \{-1, 1\}$ identifies the sign of a value. We set w to v', so obtaining a mutated model $\mathcal{M}^C_f$. The intuition of sampling around the weights values of the original controller is that we want to produce realistic DNNs that are obtainable by a good training;
  - we run $\mathcal{M}^C_f$ over each test in $\mathcal{S}$ and assess φ's satisfaction; if the safety rate $SR(\mathcal{M}^C_f, \varphi, \mathcal{S})$ is between 10% and 90%, we keep $\mathcal{M}^C_f$ as a faulty model, otherwise we discard it and sample another value; the intuition behind using this range of SR is to guarantee that all faulty benchmarks have both passing and failing tests (which is required for SBFL to be effective), and to have different types of faults. At most, we perform 20 attempts of modification of each weight w;

Table 5. Number of Faulty Benchmarks (for Each $\mathsf{SetMuts}^i_{\mathcal{M}^C\_\varphi}$ ($i \in \{1,\ldots,5\}$) and for $\mathsf{SetMuts}_{\mathcal{M}^C\_\varphi}$) Generated from the 13 Correct Benchmarks

	ACC#1_φ^1_{ACC}	ACC#1_φ^2_{ACC}	ACC#2_φ^1_{ACC}	ACC#2_φ^2_{ACC}	AFC#1_φ^1_{AFC}	AFC#1_φ^2_{AFC}	AFC#2_φ^1_{AFC}
$\mathsf{SetMuts}^1_{\mathcal{M}^C\_\varphi}$	36	39	39	59	66	49	79
$\mathsf{SetMuts}^2_{\mathcal{M}^C\_\varphi}$	179	200	195	190	164	178	147
$\mathsf{SetMuts}^3_{\mathcal{M}^C\_\varphi}$	122	186	143	182	110	147	99
$\mathsf{SetMuts}^4_{\mathcal{M}^C\_\varphi}$	77	164	93	170	68	123	65
$\mathsf{SetMuts}^5_{\mathcal{M}^C\_\varphi}$	53	123	64	151	39	87	37
$\mathsf{SetMuts}_{\mathcal{M}^C\_\varphi}$	467	712	534	752	447	584	427

	AFC#2_φ^2_{AFC}	SC#1_φ_{SC}	SC#2_φ_{SC}	SC_R_φ_{SC}	WT#1_φ_{WT}	WT#2_φ_{WT}	Total
$\mathsf{SetMuts}^1_{\mathcal{M}^C\_\varphi}$	70	12	11	20	23	71	574
$\mathsf{SetMuts}^2_{\mathcal{M}^C\_\varphi}$	169	37	44	7	177	200	1,887
$\mathsf{SetMuts}^3_{\mathcal{M}^C\_\varphi}$	124	10	13	0	136	199	1,471
$\mathsf{SetMuts}^4_{\mathcal{M}^C\_\varphi}$	91	65	3	0	85	191	1,195
$\mathsf{SetMuts}^5_{\mathcal{M}^C\_\varphi}$	61	35	57	0	56	177	940
$\mathsf{SetMuts}_{\mathcal{M}^C\_\varphi}$	515	159	128	27	477	838	6,067

—We denote by $\mathsf{SetMuts}^1_{\mathcal{M}^C\_\varphi}$ the set of all the faulty benchmarks $\mathcal{M}^C_f$ (obtained from the correct benchmark $\mathcal{M}^C\_\varphi$) containing a single fault. At most, $\mathsf{SetMuts}^1_{\mathcal{M}^C\_\varphi}$ contains as many mutants as the number of weights of $\mathcal{M}^C$.[5]

Then, we build faulty benchmarks containing higher number of faults as follows. We build faulty benchmarks $\mathsf{SetMuts}^2_{\mathcal{M}^C\_\varphi}$ having two faults by combining two single-fault benchmarks from $\mathsf{SetMuts}^1_{\mathcal{M}^C\_\varphi}$. We only keep the faulty benchmarks $\mathcal{M}^C_f$ whose safety rate $SR(\mathcal{M}^C_f, \varphi, \mathcal{S})$ is between 10% and 90%; moreover, to guarantee that the two weight mutations are both contributing to the decrease in safety (and not cancelling each other), we require that the *SR* of the combined benchmark is lower than that of the constituent single-fault faulty benchmarks. At most, we generate 200 benchmarks in $\mathsf{SetMuts}^2_{\mathcal{M}^C\_\varphi}$; however, we usually generate less than 200 benchmarks, as we do not find enough suitable combinations of single-fault benchmarks.

In a similar way, we generate benchmark set $\mathsf{SetMuts}^3_{\mathcal{M}^C\_\varphi}$ (by merging benchmarks from $\mathsf{SetMuts}^1_{\mathcal{M}^C\_\varphi}$ and $\mathsf{SetMuts}^2_{\mathcal{M}^C\_\varphi}$), benchmark set $\mathsf{SetMuts}^4_{\mathcal{M}^C\_\varphi}$ (by merging benchmarks from $\mathsf{SetMuts}^1_{\mathcal{M}^C\_\varphi}$ and $\mathsf{SetMuts}^3_{\mathcal{M}^C\_\varphi}$), and benchmark set $\mathsf{SetMuts}^5_{\mathcal{M}^C\_\varphi}$ (by merging benchmarks from $\mathsf{SetMuts}^1_{\mathcal{M}^C\_\varphi}$ and $\mathsf{SetMuts}^4_{\mathcal{M}^C\_\varphi}$).[6]

So, for each of the 13 correct benchmarks $\mathcal{M}^C\_\varphi$, we produce the set $\mathsf{SetMuts}_{\mathcal{M}^C\_\varphi} = \cup_{i\in\{1,\ldots,5\}} \mathsf{SetMuts}^i_{\mathcal{M}^C\_\varphi}$ of its faulty versions. Table 5 reports the number of generated faulty benchmarks.

[5]Note that, for some weights, in 20 attempts we were not able to change the weight in a way that ensures that the safety rate $SR(\mathcal{M}^C_f, \varphi, \mathcal{S})$ falls between 10% and 90%.

[6]Note that, by following this protocol, we were not able to build faulty benchmarks with more than one faulty weight for SC#1_φ_{SC} and SC#2_φ_{SC}, as we do not have enough benchmarks in $\mathsf{SetMuts}^1_{\mathcal{M}^C\_\varphi}$ to combine. Therefore, only for these two correct benchmarks, we produce faulty benchmarks by directly changing weights values. Moreover, for SC_R_φ_{SC}, we were not able to produce faulty benchmarks with more than two modifications for which *SR* is greater than 10%; so, we only have faulty benchmarks in $\mathsf{SetMuts}^1_{\mathcal{M}^C\_\varphi}$ and $\mathsf{SetMuts}^2_{\mathcal{M}^C\_\varphi}$ for SC_R_φ_{SC}.

Table 6. Distribution of Safety Rate *SR* across the Faulty Benchmarks

	$SR(\mathcal{M}_f^C, \varphi, \mathcal{S})$							
	[10%, 20%)	[20%, 30%)	[30%, 40%)	[40%, 50%)	[50%, 60%)	[60%, 70%)	[70%, 80%)	[80%, 90%]
$\mathsf{SetMuts}^1_{\mathcal{M}^C\_\varphi}$	12.3%	8.5%	11.0%	9.9%	10.1%	9.0%	14.1%	25.1%
$\mathsf{SetMuts}^2_{\mathcal{M}^C\_\varphi}$	4.4%	6.8%	8.5%	7.1%	11.5%	18.7%	24.8%	18.2%
$\mathsf{SetMuts}^3_{\mathcal{M}^C\_\varphi}$	14.8%	13.5%	13.3%	10.7%	16.6%	13.3%	9.9%	8.0%
$\mathsf{SetMuts}^4_{\mathcal{M}^C\_\varphi}$	20.7%	18.5%	14.1%	12.1%	14.5%	8.0%	6.2%	5.9%
$\mathsf{SetMuts}^5_{\mathcal{M}^C\_\varphi}$	23.0%	16.1%	14.5%	15.3%	13.6%	4.9%	6.1%	6.6%
$\mathsf{SetMuts}_{\mathcal{M}^C\_\varphi}$	13.7%	12.3%	11.9%	10.5%	13.6%	12.2%	13.7%	12.2%

Moreover, Table 6 shows, for different intervals of safety rate (first row), the percentage of faulty models having safety rate in that interval (second row). We observe that, overall (i.e., for $\mathsf{SetMuts}_{\mathcal{M}^C\_\varphi}$), we obtain a quite uniform distribution that covers different situations. Regarding the different subsets $\mathsf{SetMuts}^i_{\mathcal{M}^C\_\varphi}$ (with $i \in \{1, \ldots, 5\}$), we observe that, as expected, those containing more faulty weights tend to have more benchmarks with lower *SR*.

5.3 Compared Approaches

In the experiments, we assess both versions of SpectAcle, i.e., $\text{SpectAcle}_{\text{UW}}$ and $\text{SpectAcle}_{\text{W}}$. Moreover, we experiment with five different values of k (see Line 13 in Algorithm 1), that we identify as *Low*, *Mid-low*, *Mid*, *Mid-high* and *High*. The values of k depend on the number of the weights whose forward impact is greater than 0, during the execution of the entire test suite by the system. Specifically, given a faulty AI-enabled CPS $\mathcal{M}_f^C$, a specification φ, and a test suite $\mathcal{S}$, we execute $\mathcal{M}_f^C$ over $\mathcal{S}$ and record the number of the weights with a positive forward impact before τ, denoted as N_p. Then, for $\mathcal{M}_f^C$, we set k as $i \times N_p$, with $i \in \{10\%, 20\%, 30\%, 40\%, 50\%\}$, to instantiate *Low*, *Mid-low*, *Mid*, *Mid-high* and *High*. The rationale for using N_p is to filter out the weights that have never contributed to the system execution; the justification for selecting at most 50% of these weights is that higher values may introduce excessive noise into fault localisation (i.e., select too many unnecessary weights).

We compare SpectAcle with two baseline approaches. First, we compare SpectAcle with Random selection which randomly selects suspicious weights in Θ_s. In order to account for the non-determinism of Random, by following the guideline [4] of running experiments with randomised algorithms, we executed the Random approach 100 times for each faulty model, and we reported average results.

Then, to assess the importance of considering only the top-k weights, we compared SpectAcle with a simpler version of SpectAcle (called act-based) that considers all the weights that are *activated*, similarly to what is done in approaches like DeepFault [15] that, as SpectAcle, is based on SBFL; we consider a weight activated in a given inference if its forward impact is greater than 0. We created two versions of act-based: $\text{act-based}_{\text{UW}}$ and $\text{act-based}_{\text{W}}$ that use the two types of windows.

5.4 Evaluation Metrics

To assess the effectiveness of SpectAcle and of the baseline approaches (see Section 5.3), for each set of faulty benchmarks $\mathsf{SetMuts}_{\mathcal{M}^C\_\varphi}$ (see Section 5.2.2), we run each compared approach App over each $\mathcal{M}_f^C \in \mathsf{SetMuts}_{\mathcal{M}^C\_\varphi}$ and assess its **Detection Rate (DR)** defined as follows. First, we define the *recall* of an approach App over a faulty benchmark $\mathcal{M}_f^C$ as the percentage of modified

weights that have been correctly identified by App in its final results Θ_s (see Line 18 in Algorithm 1), i.e.,

$$recall(\text{App}, \mathcal{M}_f^C) = \frac{\text{\# of modified weights of } \mathcal{M}_f^C \text{ correctly identified in } \Theta_s \text{ by App}}{\text{\# of modified weights in } \mathcal{M}_f^C}$$

The *DR* is then defined as

$$DR(\text{App}, \text{SetMuts}_{\mathcal{M}^C\_\varphi}) = \frac{\sum_{\mathcal{M}_f^C \in \text{SetMuts}_{\mathcal{M}^C\_\varphi}} recall(\text{App}, \mathcal{M}_f^C)}{|\text{SetMuts}_{\mathcal{M}^C\_\varphi}|} \tag{3}$$

We calculate the *DR* for each possible value of s suspicious weights in the final results Θ_s. Subsequently, we compute the ***Area Under the Curve*** (AUC_{DR}) described by *DR* at the increase of s (we call it AUC_{DR}). The performance of the different approaches will be evaluated based on their respective AUC_{DR} values. Specifically, in order to compare the performance of the different approaches, we perform statistical analysis of their AUC_{DR} results. Given two approaches App1 and App2, we do pairwise comparison between the values of AUC_{DR} across all the faulty benchmarks and suspiciousness metrics, using the non-parametric test Wilcoxon signed-rank test. We use $\alpha = 0.05$ as significance level. If the p-value is less than α, we reject the null hypothesis that there is no significant difference. If there is a significant difference, we use Cohen's d effect size [11] to assess the strength of the significance. If $d > 0$, then App1 is better; otherwise App2 is better. We interpret the effect size using the following categories [28]: *small* if $d \in (0, 0.2)$, *medium* if $d \in [0.2, 0.8)$ and *large* if $d \geq 0.8$.

SpectAcle, as any other SBFL approach, returns false-positive results that depend on the value of the hyper-parameter s (i.e., number of weights returned in Θ_s): the higher the s, the higher the number of false-positive results. In order to assess this, given an approach App and a faulty benchmark $\mathcal{M}_f^C \in \text{SetMuts}_{\mathcal{M}^C\_\varphi}$, we first compute the *false detection* as the percentage of wrongly identified weights, i.e.,

$$falseDetection(\text{App}, \mathcal{M}_f^C) = \frac{\text{\# of modified weights of } \mathcal{M}_f^C \text{ wrongly identified in } \Theta_s \text{ by App}}{|\Theta_s|}.$$

Then, we compute *False Discovery Rate (FDR)* as

$$FDR(\text{App}, \text{SetMuts}_{\mathcal{M}^C\_\varphi}) = \frac{\sum_{\mathcal{M}_f^C \in \text{SetMuts}_{\mathcal{M}^C\_\varphi}} falseDetection(\text{App}, \mathcal{M}_f^C)}{|\text{SetMuts}_{\mathcal{M}^C\_\varphi}|}. \tag{4}$$

Note that a given level of *FDR* is unavoidable, and primarily depends on the value of s. A higher value of *FDR* will usually correspond to a higher value of *DR*; so, a user must decide s depending on the desired tradeoff between accuracy and precision of the results. Similarly to what done for *DR*, we calculate the *FDR* for each possible value of s, and compute the AUC_{DR} described by *FDR* at the increase of s (we call it AUC_{FDR}). Then, we do statistical analysis among approaches as described before. In this case, the interpretation of Cohen's d effect size is different, as a lower value of AUC_{FDR} is better. So, given two approaches App1 and App2, if $d < 0$, then App1 is better; otherwise App2 is better. We interpret the effect size using the following categories [28]: *small* if $d \in (-0.2, 0)$, *medium* if $d \in (-0.8, -0.2]$ and *large* if $d \leq -0.8$.

5.4.1 Assessment of RQs. In order to answer RQ1, we check the statistical comparison in terms of AUC_{DR} and AUC_{FDR}, between the different versions of SpectAcle and, respectively, Random and act-based.

In order to answer RQ2, we check the statistical comparison between the different versions of SpectAcle using the different values of k, i.e., *Low*, *Mid-low*, *Mid*, *Mid-high* and *High* (see Section 5.3).

In order to answer RQ3, we check the statistical comparison between SpectAcle$_{\text{uw}}$ and SpectAcle$_{\text{w}}$. Moreover, we also report, for each benchmark $\mathcal{M}^C\_\varphi$, the percentage of times (across different faulty benchmarks $\mathcal{M}^C_f$, values of k, suspiciousness metrics `SusMetr`, and values of s) that SpectAcle$_{\text{uw}}$ outperforms SpectAcle$_{\text{w}}$, that they obtain the same results and that SpectAcle$_{\text{w}}$ outperforms SpectAcle$_{\text{uw}}$.

In order to answer RQ4, we perform statistical analysis using Wilcoxon signed-rank test and Cohen's d effect size as described before. However, in this case, given two suspiciousness metrics $\texttt{SusMetr}_1$ and $\texttt{SusMetr}_2$, we compare the values of $\texttt{AUC}_{DR}$ and $\texttt{AUC}_{FDR}$ across different faulty benchmarks and settings of SpectAcle.

In order to answer RQ5, we need to select some representative faulty benchmarks to repair, as repairing all of them would take too much time. We proceed as follows. We first select four correct benchmarks: for each of the four CPSs, we select a specification and its most complex DNN controller, namely ACC#2_φ^1_{ACC}, AFC#2_φ^1_{AFC}, SC#2_φ_{SC} and WT#2_φ_{WT}. Then, for each correct benchmark $\mathcal{M}^C\_\varphi$, we select one faulty benchmark in $\mathsf{SetMuts}^1_{\mathcal{M}^C\_\varphi}$ (called $\mathcal{M}^C_{f1}$), one in $\mathsf{SetMuts}^3_{\mathcal{M}^C\_\varphi}$ (called $\mathcal{M}^C_{f3}$) and one in $\mathsf{SetMuts}^5_{\mathcal{M}^C\_\varphi}$ (called $\mathcal{M}^C_{f5}$). Therefore, we have selected 12 faulty benchmarks to repair. Then, for each $\mathcal{M}^C_f$, among all fault localisation results obtained with the different versions of SpectAcle, we select those that allow identifying all the modified weights with the minimum value of top s. Then, we run a search-based DNN repair approach that has been recently proposed for AI-enabled CPSs [36]; the approach searches for alternative values of some weights of the DNN controller, with the goal of maximising the number of passing tests in a test suite $\mathcal{S}$. As search variables, we use the weights Θ_s that have been identified by SpectAcle; as test suite $\mathcal{S}$, we use the same used by SpectAcle (see Section 5.2.2). We use a population size of 50 individuals, and run the approach for 50 generations. Finally, we check the safety rate *SR* obtained by the final repaired model.

In order to answer RQ6, we select, for each of the 13 correct benchmarks $\mathcal{M}^C\_\varphi$, five faulty benchmarks, one for each set of mutants $\mathsf{SetMuts}^i_{\mathcal{M}^C\_\varphi}$ ($i \in \{1, \ldots, 5\}$). Then, we record the time for executing the test suite (i.e., the simulation of the tests), and the time spent for doing fault localisation. We report the time for doing fault localisation in terms of time spent for calculating the forward impact and for calculating the execution spectrum and the suspiciousness score.

5.5 Software and Hardware Specifications

All subject CPSs are implemented in Simulink with dependencies on MATLAB *Deep Learning Toolbox*. The experiments were conducted on a server with a 3.3 GHz Intel i9-10940X CPU, 64G RAM, and two NVIDIA RTX A6000 GPUs, 10 threads.

6 Experimental Evaluation

In this section, we present the experimental results. Before answering the RQs, we first provide a visual representation of the results. Figure 6 illustrates the overall effectiveness of SpectAcle and of the compared approaches (see Section 5.3) on the 13 groups of faulty benchmarks (see Section 5.2), using Kulczynski2 as the suspiciousness metric.[7] Each figure represents, for a given benchmark $\mathcal{M}^C\_\varphi$, the *DR* (see Equation 3) of the approaches, i.e., their ability in identifying the faulty weights of the faulty versions $\mathsf{SetMuts}_{\mathcal{M}^C\_\varphi}$. The x-axis represents the percentage of weights examined in the final ranking of suspicious weights; this is related to the size s of the set of returned suspicious weights Θ_s in Line 18 of Algorithm 1. The y-axis represents the $DR(\text{App}, \mathsf{SetMuts}_{\mathcal{M}^C\_\varphi})$. We report

[7] We selected the Kulczynski2 metric, as it is the best metric according to the results of RQ4 in Section 6.4.

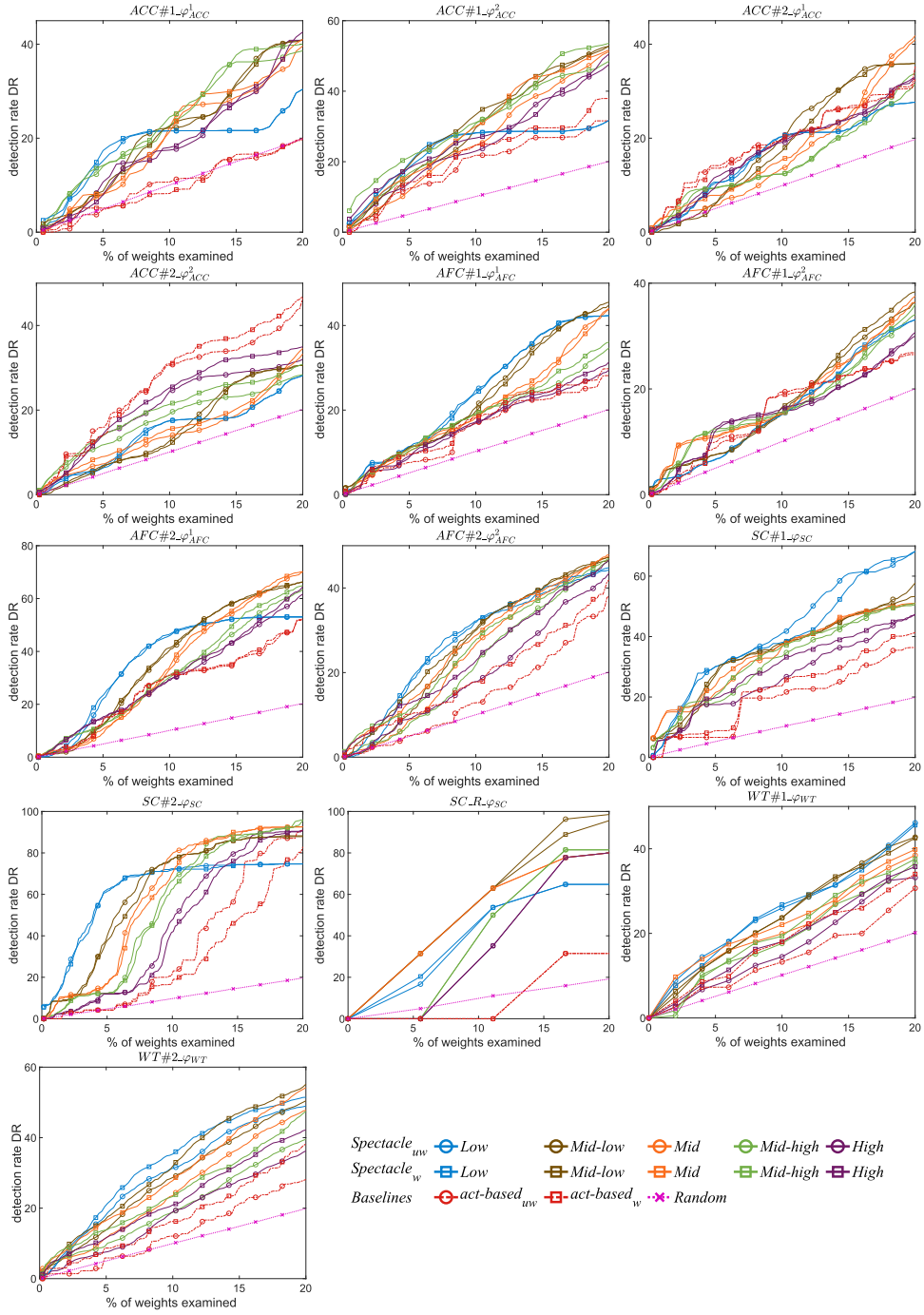

Fig. 6. Effectiveness of the compared approaches (in terms of *DR*) on the 13 groups of faulty benchmarks (using Kulczynski2 as suspiciousness metric `SusMetr`).

Table 7. Effectiveness of the Compared Approaches in Terms of $\texttt{AUC}_{DR}$ with $s \leq 20\%$, Using Kulczynski2 as Suspiciousness Metric `SusMetr`

	SpectAcle$_{uw}$					act-based$_{uw}$	SpectAcle$_{w}$					act-based$_{w}$	Random
	Low	*Mid-Low*	*Mid*	*Mid-High*	*High*		*Low*	*Mid-Low*	*Mid*	*Mid-High*	*High*		
ACC#1_φ^1_{ACC}	0.1806	0.2105	0.1930	**0.2297**	0.1882	0.0994	0.1847	0.2090	0.2028	**0.2377**	0.1855	0.0966	0.0998
ACC#1_φ^2_{ACC}	0.2301	**0.2843**	0.2568	0.2813	0.2463	0.1849	0.2335	0.3074	0.2924	**0.3179**	0.2662	0.2102	0.1008
ACC#2_φ^1_{ACC}	0.1660	0.1939	0.1650	0.1528	0.1752	**0.1937**	0.1648	0.1837	0.1736	0.1467	0.1752	**0.1891**	0.0993
ACC#2_φ^2_{ACC}	0.1448	0.1536	0.1424	0.1744	0.2052	**0.2575**	0.1429	0.1494	0.1523	0.1947	0.2226	**0.2667**	0.1008
AFC#1_φ^1_{AFC}	**0.2400**	0.2274	0.1972	0.1800	0.1598	0.1504	**0.2383**	0.2237	0.1996	0.1874	0.1674	0.1660	0.1017
AFC#1_φ^2_{AFC}	0.1612	0.1736	**0.1786**	0.1690	0.1567	0.1578	0.1620	0.1793	**0.1819**	0.1730	0.1577	0.1547	0.1004
AFC#2_φ^1_{AFC}	**0.3653**	0.3619	0.3326	0.3056	0.2933	0.2573	**0.3696**	0.3665	0.3433	0.3175	0.3022	0.2612	0.1010
AFC#2_φ^2_{AFC}	**0.2692**	0.2596	0.2408	0.2199	0.1953	0.1433	**0.2800**	0.2751	0.2658	0.2532	0.2372	0.1885	0.1005
SC#1_φ_{SC}	**0.4117**	0.3451	0.3372	0.3133	0.2620	0.1901	**0.3985**	0.3464	0.3454	0.3300	0.2915	0.2211	0.1008
SC#2_φ_{SC}	**0.6210**	0.6201	0.5957	0.5415	0.4571	0.3436	**0.6153**	0.6089	0.5785	0.5230	0.4273	0.2702	0.0973
SC_R_φ_{SC}	0.4424	**0.6250**	0.5478	0.4784	0.4270	0.1312	0.4475	**0.6044**	0.5478	0.4784	0.4270	0.1312	0.1114
WT#1_φ_{WT}	**0.2457**	0.2338	0.2058	0.1846	0.1609	0.1361	**0.2429**	0.2331	0.2208	0.1995	0.1821	0.1745	0.1001
WT#2_φ_{WT}	**0.2962**	0.2755	0.2435	0.1992	0.1802	0.1268	**0.3244**	0.3137	0.2857	0.2408	0.2206	0.1700	0.0995

The higher the value, the better (the best approach and second best approach are highlighted in gray and light gray, respectively).

results with percentage of examined weights ranging from 0 to 20%, as it is not reasonable to have a fault localisation approach that considers too many weights. Table 7 reports the corresponding values of $\texttt{AUC}_{DR}$.[8]

In Figure 6, the higher the percentage of weights considered (i.e., the value of s), the higher the *recall*, i.e., the probability that an approach returns in Θ_s the faulty weights (see Line 18 in Algorithm 1). However, a higher s also means a higher *false detection*, i.e., a higher number of non-faulty weights; in this case, the effort required to identify the real fault among all weights in Θ_s increases (e.g., a repair approach taking in input fault localisation results may need to consider too many weights for repair). In order to assess this effect, Figure 7 reports, for each benchmark $\mathcal{M}^C\_\varphi$ and approach App, the *FDR* (App, $\text{SetMuts}_{\mathcal{M}^C\_\varphi}$) (see Equation 4) for increasing values of s (up to 20% of weights).[9] As explained in Section 5.4, a given level of *FDR* is unavoidable; the dashed black line in the figures represents the minimum value of *FDR* that is achievable. We notice that, in general, the *FDR* decreases at the beginning of the plot, until it reaches the value of s that allows to obtain the best tradeoff between the correctly identified and the wrongly identified weights; after that point, *FDR* inevitably increases, as any additional weight returned in Θ_s is most likely not faulty. Table 8 reports the corresponding values of $\texttt{AUC}_{FDR}$.

In the following, we answer the RQs reported in Section 5.1.

6.1 Answer to RQ1

In this RQ, we are interested in assessing whether the proposed approach is better than the baselines approaches in detecting the faulty weights. To do this, for each pair of approaches, we conducted pairwise comparisons between the baseline approaches and all versions of SpectAcle across all the

[8]In the supplementary Web site [38], we report *DR* plots and $\texttt{AUC}_{DR}$ tables for all the suspiciousness metrics, for $s \leq 20\%$ and $s \leq 100\%$.

[9]In the supplementary Web site [38], we report *FDR* plots and $\texttt{AUC}_{FDR}$ tables for all the suspiciousness metrics, for $s \leq 20\%$ and $s \leq 100\%$.

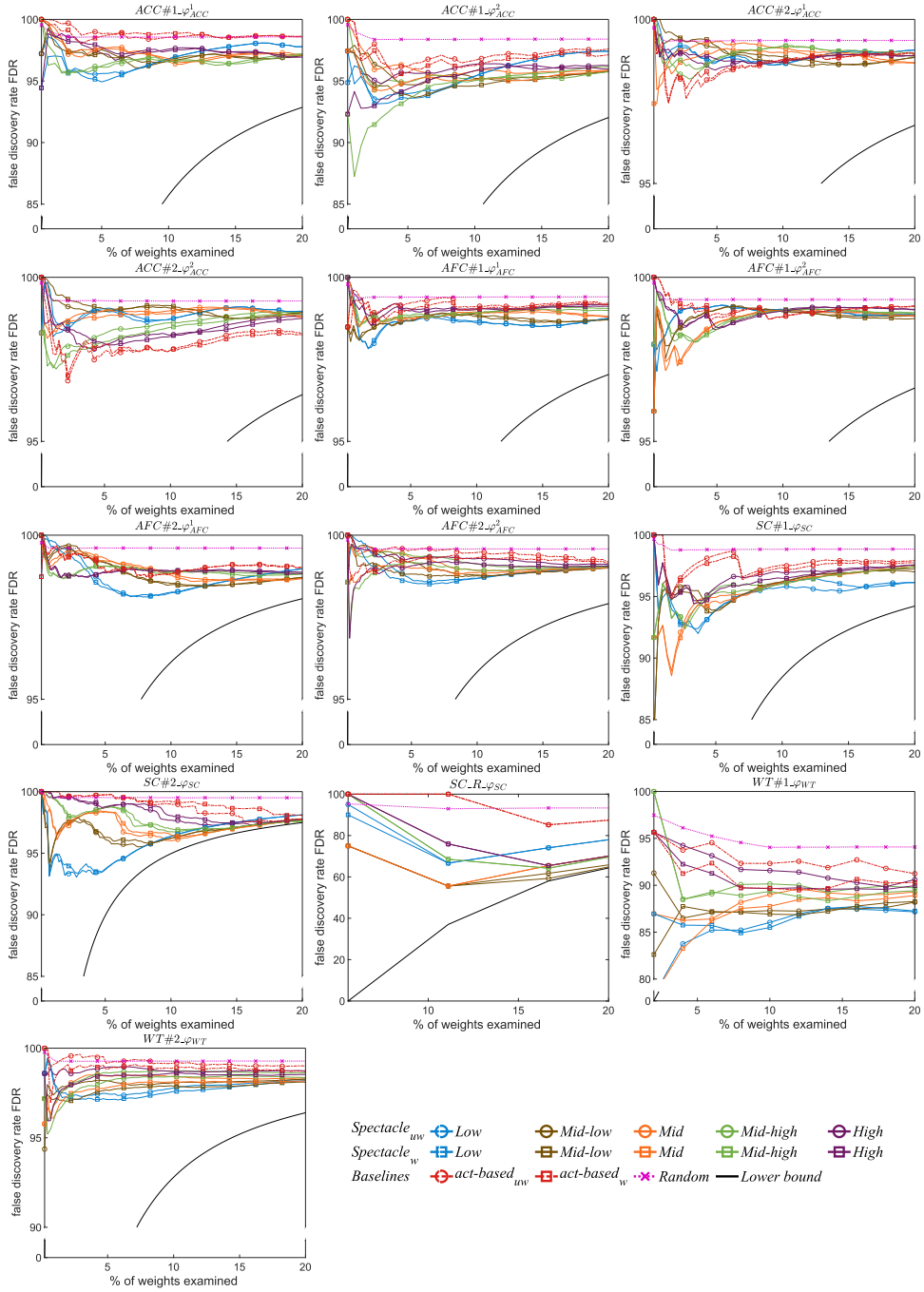

Fig. 7. Effectiveness of the compared approaches (in terms of *FDR*) on the 13 groups of faulty benchmarks (using Kulczynski2 as suspiciousness metric `SusMetr`).

Table 8. Effectiveness of the Compared Approaches in Terms of $\texttt{AUC}_{FDR}$ with $s \leq 20\%$, Using Kulczynski2 as Suspiciousness Metric `SusMetr`

	SpectAcle$_{uw}$					act-based$_{uw}$	SpectAcle$_{w}$					act-based$_{w}$	Random	Theoretical Lower Bound
	Low	*Mid-Low*	*Mid*	*Mid-High*	*High*		*Low*	*Mid-Low*	*Mid*	*Mid-High*	*High*			
ACC#1_φ^1_{ACC}	0.9214	0.9219	0.9243	**0.9171**	0.9262	0.9383	0.9199	0.9215	0.9233	**0.9172**	0.9266	0.9380	0.9366	0.7479
ACC#1_φ^2_{ACC}	0.9082	0.9071	0.9112	**0.9067**	0.9116	0.9220	0.9073	0.9015	0.9043	**0.8964**	0.9055	0.9185	0.9349	0.7276
ACC#2_φ^1_{ACC}	0.9668	0.9667	0.9679	0.9672	0.9661	**0.9643**	0.9672	0.9676	0.9668	0.9678	0.9665	**0.9648**	0.9715	0.8604
ACC#2_φ^2_{ACC}	0.9667	0.9677	0.9673	0.9632	0.9622	**0.9580**	0.9672	0.9684	0.9666	0.9618	0.9616	**0.9582**	0.9708	0.8497
AFC#1_φ^1_{AFC}	**0.9640**	0.9648	0.9666	0.9675	0.9689	0.9699	**0.9641**	0.9651	0.9665	0.9670	0.9682	0.9688	0.9719	0.8697
AFC#1_φ^2_{AFC}	0.9672	0.9665	**0.9651**	0.9661	0.9673	0.9681	0.9673	0.9663	**0.9651**	0.9659	0.9672	0.9684	0.9712	0.8560
AFC#2_φ^1_{AFC}	**0.9724**	0.9736	0.9748	0.9744	0.9744	0.9760	**0.9720**	0.9733	0.9743	0.9742	0.9743	0.9759	0.9813	0.9069
AFC#2_φ^2_{AFC}	**0.9747**	0.9757	0.9765	0.9773	0.9781	0.9799	**0.9737**	0.9742	0.9746	0.9749	0.9756	0.9775	0.9811	0.9012
SC#1_φ_{SC}	**0.9195**	0.9287	0.9243	0.9283	0.9347	0.9442	**0.9207**	0.9279	0.9232	0.9269	0.9321	0.9417	0.9555	0.7863
SC#2_φ_{SC}	**0.9462**	0.9529	0.9569	0.9616	0.9673	0.9732	**0.9469**	0.9538	0.9582	0.9628	0.9690	0.9764	0.9804	0.8874
SC_R_φ_{SC}	0.4102	**0.3369**	0.3605	0.3751	0.3888	0.4990	0.4102	**0.3438**	0.3605	0.3751	0.3888	0.4990	0.5179	0.3078
WT#1_φ_{WT}	**0.6912**	0.6987	0.7082	0.7173	0.7311	0.7407	**0.6921**	0.6992	0.7016	0.7117	0.7202	0.7226	0.7553	0.3738
WT#2_φ_{WT}	**0.9562**	0.9590	0.9609	0.9631	0.9656	0.9700	**0.9545**	0.9558	0.9574	0.9596	0.9618	0.9665	0.9708	0.8490

The lower the value, the better (the best approach and second best approach are highlighted in gray and light gray, respectively).

Table 9. RQ1, RQ2, RQ3–Statistical Comparison Between Each Pair of Approaches in Terms of $\texttt{AUC}_{DR}$

		SpectAcle$_{uw}$					act-based$_{uw}$	SpectAcle$_{w}$					act-based$_{w}$	Random
		Low	*Mid-Low*	*Mid*	*Mid-High*	*High*		*Low*	*Mid-Low*	*Mid*	*Mid-High*	*High*		
SpectAcle$_{uw}$	*Low*	≡	✗	✓✓	✓✓	✓✓	✓✓	✗✗	✗✗	✓	✓✓	✓✓	✓✓	✓✓✓
	Mid-low	✓	≡	✓✓	✓✓	✓✓	✓✓	✗	✗✗	✓✓	✓✓	✓✓	✓✓	✓✓✓
	Mid	✗✗	✗✗	≡	✓✓	✓✓	✓✓	✗✗	✗✗	✗	✓✓	✓✓	✓✓	✓✓✓
	Mid-high	✗✗	✗✗	✗✗	≡	✓✓	✓✓	✗✗	✗✗	✗✗	✓	✓✓	✓✓	✓✓✓
	High	✗✗	✗✗	✗✗	✗✗	≡	✓✓	✗✗	✗✗	✗✗	✗	✓	✓✓	✓✓✓
act-based$_{uw}$		✗✗	✗✗	✗✗	✗✗	✗✗	≡	✗✗	✗✗	✗✗	✗✗	✗✗	✗	✓✓✓
SpectAcle$_{w}$	*Low*	✓✓	✓	✓✓	✓✓	✓✓	✓✓	≡	✗✗	✓✓	✓✓	✓✓	✓✓	✓✓✓
	Mid-low	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	≡	✓✓	✓✓	✓✓	✓✓	✓✓✓
	Mid	✗	✗✗	✓	✓✓	✓✓	✓✓	✗✗	✗✗	≡	✓✓	✓✓	✓✓	✓✓✓
	Mid-high	✗✗	✗✗	✗✗	✗	✓	✓✓	✗✗	✗✗	✗✗	≡	✓✓	✓✓	✓✓✓
	High	✗✗	✗✗	✗✗	✗✗	✗	✓✓	✗✗	✗✗	✗✗	✗✗	≡	✓✓	✓✓✓
act-based$_{w}$		✗✗	✗✗	✗✗	✗✗	✗✗	✓	✗✗	✗✗	✗✗	✗✗	✗✗	≡	✓✓✓
Random		✗✗✗	✗✗✗	✗✗✗	✗✗✗	✗✗✗	✗✗✗	✗✗✗	✗✗✗	✗✗✗	✗✗✗	✗✗✗	✗✗✗	≡

≡: there is no difference between the two approaches. ✓, ✓✓, ✓✓✓: the approach on the rows is significantly better than the approach on the column with strength *small*, *medium* and *large*. ✗, ✗✗, ✗✗✗: the approach on the row is significantly worse than the approach on the column with strength *small*, *medium* and *large*.

benchmarks and different suspiciousness metrics in terms of $\texttt{AUC}_{DR}$ and $\texttt{AUC}_{FDR}$, using the Wilcoxon signed-rank test and Cohen's *d* effect size (see Section 5.4). Table 9 reports the results for $\texttt{AUC}_{DR}$. Table 10, instead, reports the results for $\texttt{AUC}_{FDR}$. First of all, by comparing the results in Tables 9 and 10, we observe that the results are almost always consistent: whenever an approach is better in terms of $\texttt{AUC}_{DR}$, it is also better in terms of $\texttt{AUC}_{FDR}$, with similar strength of the effect size.[10] So, for simplicity, in this and in the following RQs, we will focus our analysis on the results of $\texttt{AUC}_{DR}$, as also considering $\texttt{AUC}_{FDR}$ does not provide any additional insight.

[10]The only exceptions are the comparison between SpectAcle$_{uw}$, *Low* and SpectAcle$_{w}$, *Mid-low*, and between SpectAcle$_{uw}$, *Mid-high* and SpectAcle$_{w}$, *Mid-high*, in which the result changes between $\texttt{AUC}_{DR}$ and $\texttt{AUC}_{FDR}$. However, in both cases, the strength of the effect size is *small*, so not too significant.

Table 10. RQ1, RQ2, RQ3—Statistical Comparison Between Each Pair of Approaches in Terms of $\texttt{AUC}_{FDR}$

		SpectAcle$_{UW}$					ACT-based$_{UW}$	SpectAcle$_{W}$					ACT-based$_{W}$	Random
		Low	*Mid-Low*	*Mid*	*Mid-High*	*High*		*Low*	*Mid-Low*	*Mid*	*Mid-High*	*High*		
SpectAcle$_{UW}$	*Low*	≡	✗✗	✓	✓	✓✓	✓✓	✗✗	✗✗	✗	✓✓	✓✓	✓✓	✓✓✓
	Mid-low	✓✓	≡	✓✓	✓✓	✓✓	✓✓	✓	✗✗	✓✓	✓✓	✓✓	✓✓	✓✓
	Mid	✗	✗✗	≡	✓✓	✓✓	✓✓	✗	✗✗	✗	✓✓	✓✓	✓✓	✓✓
	Mid-high	✗	✗✗	✗✗	≡	✓✓✓	✓✓	✗	✗✗	✗	✓✓	✓✓	✓✓	✓✓
	High	✗✗	✗✗	✗✗	✗✗✗	≡	✓✓	✗✗	✗✗	✗✗	✓	✓	✓✓	✓✓
ACT-based$_{UW}$		✗✗	✗✗	✗✗	✗✗	✗✗	≡	✗✗	✗✗	✗✗	✗✗	✗✗	✗✗	✓✓✓
SpectAcle$_{W}$	*Low*	✓✓	✗	✓	✓	✓✓	✓✓	≡	✗✗	✓	✓✓	✓✓	✓✓	✓✓✓
	Mid-low	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	≡	✓✓	✓✓	✓✓	✓✓	✓✓
	Mid	✓	✗✗	✓	✓	✓✓	✓✓	✗	✗✗	≡	✓✓	✓✓	✓✓	✓✓
	Mid-high	✗✗	✗✗	✗✗	✗✗	✗	✓✓	✗✗	✗✗	✗✗	≡	✓✓	✓✓	✓✓
	High	✗✗	✗✗	✗✗	✗✗	✗	✓✓	✗✗	✗✗	✗✗	✗✗	≡	✓	✓✓
ACT-based$_{W}$		✗✗	✗✗	✗✗	✗✗	✗✗	✓✓	✗✗	✗✗	✗✗	✗✗	✗	≡	✓✓✓
Random		✗✗✗	✗✗	✗✗	✗✗	✗✗	✗✗✗	✗✗✗	✗✗	✗✗	✗✗	✗✗	✗✗✗	≡

≡: there is no difference between the two approaches. ✓, ✓✓, ✓✓✓: the approach on the rows is significantly better than the approach on the column with strength *small*, *medium* and *large*. ✗, ✗✗, ✗✗✗: the approach on the row is significantly worse than the approach on the column with strength *small*, *medium* and *large*.

By looking at Table 9, we observe that any version of SpectAcle is always better than Random, with *large* strength. This demonstrates the effectiveness of the proposed fault localisation in finding faults.

Moreover, we observe that any version of SpectAcle is better than the two versions of the baseline approach ACT-based (ACT-based$_{UW}$ and ACT-based$_{W}$), always with *medium* strength. This shows that the top-k criterion adopted by SpectAcle is more effective than the *activation* criterion adopted by ACT-based for the construction of the spectrum.

Answer to RQ1: SpectAcle can effectively localise the faulty DNN weights in AI-enabled CPSs and performs better than the two baseline approaches Random and ACT-based.

6.2 Answer to RQ2

In this RQ, we are interested in assessing which value of hyper-parameter k provides the best results. As explained in Section 5.3, we experimented with five different values of different magnitudes, i.e., *Low*, *Mid-low*, *Mid*, *Mid-high* and *High*.

In Table 9, we check the comparisons of the different versions of SpectAcle$_{UW}$ among themselves, and those of the versions of SpectAcle$_{W}$ among themselves. We notice that, for both SpectAcle$_{UW}$ and SpectAcle$_{W}$, *Low* and *Mid-low* outperform other top-k values, which indicates that a too large top-k value is not suitable for an effective fault localisation. *Mid-low* is also always better than *Low*, once with *small* effect size (for SpectAcle$_{UW}$) and once with *medium* effect size (for SpectAcle$_{W}$); still, we think that also *Mid-low* could be effective in some cases.

These results show that SpectAcle can obtain the best performance without selecting too many unnecessary weights.

Answer to RQ2: Among the five top-k value categories, *Low* and *Mid-low* can localise faulty weights more effectively.

6.3 Answer to RQ3

In this RQ, we are interested in assessing the effectiveness of SpectAcle$_{UW}$ and SpectAcle$_{W}$, i.e., the contribution of using a constant window or the Hamming window (see Section 4.3.1). From Table 9, we observe that SpectAcle$_{W}$ with *Low* and *Mid-low* is always better than any version of

Table 11. RQ3—Comparison between SpectAcle$_{UW}$ and SpectAcle$_{W}$ (Better Approach in Gray)

	UW	=	W		UW	=	W
ACC#1_φ^1_{ACC}	27.2%	10.1%	**62.7%**	ACC#1_φ^2_{ACC}	11.0%	8.6%	**80.4%**
ACC#2_φ^1_{ACC}	**50.4%**	10.5%	39.1%	ACC#2_φ^2_{ACC}	34.0%	10.8%	**55.2%**
AFC#1_φ^1_{AFC}	29.2%	4.2%	**66.6%**	AFC#1_φ^2_{AFC}	22.4%	0.5%	**77.0%**
AFC#2_φ^1_{AFC}	30.5%	1.5%	**68.0%**	AFC#2_φ^2_{AFC}	10.3%	0.6%	**89.1%**
SC#1_φ_{SC}	**51.3%**	1.6%	47.1%	SC#2_φ_{SC}	**63.1%**	10.7%	26.2%
SC_R_φ_{SC}	**27.0%**	48.0%	25.0%	WT#1_φ_{WT}	23.2%	0.8%	**76.0%**
WT#2_φ_{WT}	9.9%	2.1%	**88.0%**	All	31.8%	5.3%	**62.9%**

SpectAcle$_{UW}$, almost always with strength *medium* (except for one case in which the strength is *small*).

To better analyse the results, in Table 11, we report, for each benchmark $\mathcal{M}^C$_φ, the percentage of times (across different values of k, suspiciousness metrics, and values of s) that SpectAcle$_{UW}$ is better than SpectAcle$_{W}$ (column UW), that SpectAcle$_{W}$ is better than SpectAcle$_{UW}$ (column W) and that they are the same (column =). We observe that SpectAcle$_{W}$ outperforms SpectAcle$_{UW}$ on 9/13 of the benchmarks, to varying degrees, ranging from a 21.2% difference (ACC#2_φ^2_{ACC}) to a 78.8% (AFC#2_φ^2_{AFC}). By considering all the benchmarks together (line 'All'), SpectAcle$_{W}$ is better in 62.9% of the cases (i.e., same benchmark, same k, same suspiciousness metric, and same s), while SpectAcle$_{UW}$ is better in only 31.8% of the cases.

These results confirm our intuition that weighting more the weights that are relevant close to the violation episode (as done by the Hamming window used in SpectAcle$_{W}$) is better than considering all the inferences equally (as done by the constant window used in SpectAcle$_{UW}$).

Answer to RQ3: SpectAcle$_{W}$ outperforms SpectAcle$_{UW}$ in most cases, which confirms that the Hamming window can enhance the effectiveness of SpectAcle.

6.4 Answer to RQ4

In this RQ, we are interested in assessing the influence of the different suspiciousness metrics on the effectiveness of SpectAcle. Similarly to the previous RQs, we performed pairwise comparisons between the five suspiciousness metrics across all the benchmarks and different hyper-parameters in terms of AUC$_{DR}$ and AUC$_{FDR}$, using the Wilcoxon signed-rank test and the Cohen's d effect size (see Section 5.4). Results are reported in Tables 12 and 13, respectively.

First of all, by comparing the two tables, we observe that the results are almost always consistent (except for the comparison of D* and Jaccard), i.e., whenever an approach is better of another in terms of AUC$_{DR}$, it is also better in terms of AUC$_{FDR}$, with almost the same strength of effect size. So, for simplicity, in the following we will focus on the description of the results of AUC$_{DR}$.

From Table 12, we observe that Tarantula is the worst metric, never better than any other metric. This result seems to be in agreement with reports about the low effectiveness of Tarantula in SBFL for classic code [29, 46]. Kulczynski2, instead, is the best metric, always better than the other metrics (half of the times with strength *medium*). A possible explanation is that, unlike Tarantula, Kulczynski2 gives more importance to *ef* (i.e., the weights that are relevant in failure test cases) which has been shown to be important in SBFL. On the other hand, Ochiai and D*, that also focus

Table 12. RQ4—Statistical Comparison between Each Pair of Suspiciousness Metrics in Terms of $\texttt{AUC}_{DR}$

	Tarantula	Ochiai	D*	Jaccard	Kulczynski2
Tarantula	≡	✗✗	✗✗	✗✗	✗✗
Ochiai	✓✓	≡	✓✓	✓	✗
D*	✓✓	✗✗	≡	✗	✗✗
Jaccard	✓✓	✗	✓	≡	✗
Kulczynski2	✓✓	✓	✓✓	✓	≡

≡: there is no difference between the two metrics. ✓, ✓✓, ✓✓✓: the metric on the rows is significantly better than the metric on the column with strength *small*, *medium* and *large*. ✗, ✗✗, ✗✗✗: the metric on the row is significantly worse than the metric on the column with strength *small*, *medium* and *large*.

Table 13. RQ4—Statistical Comparison between Each Pair of Suspiciousness Metrics in Terms of $\texttt{AUC}_{FDR}$

	Tarantula	Ochiai	D*	Jaccard	Kulczynski2
Tarantula	≡	✗✗	✗✗	✗✗	✗✗
Ochiai	✓✓	≡	✓	✓	✗
D*	✓✓	✗	≡	✓	✗✗
Jaccard	✓✓	✗	✗	≡	✗
Kulczynski2	✓✓	✓	✓✓	✓	≡

≡: there is no difference between the two metrics. ✓, ✓✓, ✓✓✓: the metric on the rows is significantly better than the metric on the column with strength *small*, *medium* and *large*. ✗, ✗✗, ✗✗✗: the metric on the row is significantly worse than the metric on the column with strength *small*, *medium* and *large*.

on *ef*, do not obtain such good results. The reason is that they may focus too much on *ef* [59]; while this is very important in fault localisation for code, it can be excessive for DNNs. These results combined show that Kulczynski2 provides the right balance between *ef* and *ep*.

Answer to RQ4: Among the five suspiciousness metrics, Kulczynski2 is the most effective, while Tarantula is always the worst.

6.5 Answer to RQ5

In this RQ, we are interested in assessing whether the weights identified by SpectAcle can be used to improve the performance of the AI-enabled CPS. As explained in Section 5.4.1, we select three faulty benchmarks ($\mathcal{M}^C_{f1}$, $\mathcal{M}^C_{f3}$, and $\mathcal{M}^C_{f5}$) for each of the four CPSs (for each CPS, we select the one with the most complex DNN controller), and we repair it using the search-based approach proposed in [36], using the suspicious weights returned by SpectAcle as search variables. Table 14 reports the experimental results. The table reports the values of the safety rate *SR* on the original faulty AI-enabled CPS *before repair*, and on the AI-enabled CPS repaired using the weights identified by SpectAcle. We observe that the repair process is almost always able to completely repair the system (i.e., *SR* = 100%); the only exception is $\mathcal{M}^C_{f5}$ of AFC#2_φ^1_{AFC} for which the final *SR* is 82%. This shows that SpectAcle can indeed identify weights that are responsible for misbehaviour.

Table 14. RQ5—Results of Repair of AI-Enabled CPSs Using the Weights Identified by SpectAcle

	ACC#2_φ^1_{ACC}			AFC#2_φ^1_{AFC}			SC#2_φ_{SC}			WT#2_φ_{WT}		
	$\mathcal{M}^C_{f1}$	$\mathcal{M}^C_{f3}$	$\mathcal{M}^C_{f5}$	$\mathcal{M}^C_{f1}$	$\mathcal{M}^C_{f3}$	$\mathcal{M}^C_{f5}$	$\mathcal{M}^C_{f1}$	$\mathcal{M}^C_{f3}$	$\mathcal{M}^C_{f5}$	$\mathcal{M}^C_{f1}$	$\mathcal{M}^C_{f3}$	$\mathcal{M}^C_{f5}$
SR before repair	73	11	55	47	21	53	16	74	82	68	64	44
SR after repair	100	100	100	100	100	82	100	100	100	100	100	100

Table 15. RQ6—Computational Cost of SpectAcle (Secs.)

	$\mathcal{S}$ Execution	Fault Localisation	
		Forward Impact	Execution Spectrum and Suspiciousness Score
ACC#1_φ^1_{ACC}	303.1	216.4	28.5
ACC#1_φ^2_{ACC}	306.7	246.2	41.8
ACC#2_φ^1_{ACC}	330.9	281.5	44.7
ACC#2_φ^2_{ACC}	340.7	259.9	61.3
AFC#1_φ^1_{AFC}	339.6	135.7	37.8
AFC#1_φ^2_{AFC}	343.2	161.8	46.8
AFC#2_φ^1_{AFC}	389.7	226.2	45.1
AFC#2_φ^2_{AFC}	382.4	242.1	55.9
SC#1_φ_{SC}	324.1	257.2	47.5
SC#2_φ_{SC}	389.0	256.6	83.0
SC_R_φ_{SC}	29.8	20.2	14.5
WT#1_φ_{WT}	234.5	76.7	9.3
WT#2_φ_{WT}	337.3	82.7	27.2
Avg.	311.6	189.5	41.8

Answer to RQ5: SpectAcle identify weights that are indeed responsible for the misbehaviour of the AI-enabled CPS and, therefore, can be targeted when improving the system performance by, e.g., search-based repair.

6.6 Answer to RQ6

In this RQ, we are interested in assessing the computational cost of SpectAcle. Table 15 reports, for each correct benchmark $\mathcal{M}^C\_\varphi$, the average time taken by SpectAcle across five faulty benchmarks of $\mathcal{M}^C\_\varphi$. Specifically, the table reports the time spent in executing the test suite $\mathcal{S}$, and the time spent in fault localisation. This latter is further divided into the time spent in calculating the forward impact, and the time spent in calculating the execution spectrum and the suspiciousness score. We observe that most of time is taken by the execution of the test suite $\mathcal{S}$, taking on average 311.6 seconds; this time will increase proportionally with the size $\mathcal{S}$. Regarding the actual time taken by fault localisation, we observe that the forward impact calculation is the most costly one (189.5 seconds on average). On the other hand, the computation of the spectrum and of the suspiciousness score is faster (41.8 seconds on average). Regarding the different benchmarks, we observe that those with smaller DNN controllers (such as SC_R, WT#1, and WT#2) tend to take less time in fault localisation; this is reasonable, as they contain less weights that need to be checked. On the

other hand, the execution time of fault localisation also depends on the number of failing tests; indeed, on failing tests, fewer inferences are checked (i.e., those till the specification violation) in comparison to the passing tests, so reducing time.

Answer to RQ6: The most computationally expensive part of SpectAcle is the execution of the test suite, followed by computation of the forward impact.

7 Threats to validity

Here, we discuss some threats that can affect the validity of the approach, following the typical classification of *construct validity*, *conclusion validity*, *internal validity* and *external validity* [54].

7.1 Construct Validity

The metrics used in the assessment of the approach could not reflect the object of the investigation. In our case, since we are interested in measuring the effectiveness of SpectAcle, we mutate some weights of the target DNN and examine whether those mutated weights can be localised, i.e., ranking at the top s. In this sense, the results largely rely on the value s and the mutations we made. To mitigate this threat, we examine the performance on different settings of top s, as done in other DNN fault localisation works [15]. Moreover, SpectAcle requires the setting of a threshold k to decide whether a weight can be counted as executed or not. A too small k could results in considering too few weights as executed. To mitigate this threat, we tried different top-k values and checked the performance of SpectAcle.

7.2 Conclusion Validity

The random behaviour of the Random approach can affect the final result. Following the guideline of Arcuri and Briand [4], we executed the Random approach 100 times for each faulty model, and we reported average results. SpectAcle and act-based, instead, are deterministic and there is no need to repeat them multiple times.

7.3 Internal Validity

A threat of this type could be that the experimental results are obtained by chance. For example, an *instrumentation* threat (e.g., a faulty implementation) may affect the execution of the experiments. To mitigate this threat, we have extensively tested the implementation of all the compared approaches.

7.4 External Validity

One such a threat is related to the generalisability of SpectAcle. To mitigate this issue, we conduct our work with a general model of AI-enabled CPS that contains a black-box physical plant and a DNN controller. Moreover, we experimented with different types of controllers, i.e., feed-forward neural networks and recurrent neural networks.

8 Related Work

In this section, we discuss related work from two perspectives: fault localisation approaches for single-inference DNNs (Section 8.1) and fault localisation approaches for classic CPSs (Section 8.1).

8.1 DNN Fault Localisation

Different works have been proposed for DNN fault localisation.

A first group of approaches target neurons and weights of the DNN model in the localisation. Ma et al. [39] introduce *MODE*, which utilises state-differential analysis between misclassification and correct classification to identify faulty neurons. Eniser et al. [15] propose *DeepFault* that

localises suspicious neurons, by building their spectra based on the activation status and using SBFL metrics (Tarantula, Ochiai and D*) to assign a suspiciousness score. Duran et al. [14] introduce *DeepFaultGr* that extends *DeepFault*, and show that doing fault localisation for different misclassifications together can produce sub-optimal results, because of the masking effect between different misclassifications. Sohn et al. [47] propose *Arachne* for repair of DNN models, in which they also handle the fault localisation problem using gradient loss and forward impact to identify the faulty neuron weights. With the previous approaches, SpectAcle shares the fact that its target weights (as *Arachne*) and that uses SBFL (as *DeepFault* and *DeepFaultGr*). However, there are three main differences: they rely on ground-truth labels, while SpectAcle relies on the system level specification; they consider single inferences of the DNN, while SpectAcle considers sequences of inferences; SpectAcle gives more or less importance to an inference in the spectrum depending on the time when the inference occurred.

A second group of approaches, instead, target faults in the configuration of the **Deep Learning (DL**) program, like wrong activation function or missing/redundant/wrong layer. For example, Wardat et al. [53] propose *DeepLocalize* that targets the faults in DNN model structures (e.g., faulty hyper-parameter), by analysing the traces produced by the DNN over training. Wardat et al. [52] propose *DeepDiagnosis* that extends *DeepLocalize* to detect more faults during training and, moreover, it also suggests possible actions to fix the fault. Cao et al. [10] introduce *DeepFD*, that, similarly to *DeepLocalize* and *DeepDiagnosis*, targets faults in DL programs. However, *DeepFD* does not use a fixed set of detection rules, but adopts a learning-based fault localisation framework which monitors the DNN model training and infers the suspicious fault types. Ghanbari et al. [18] propose *deepmufl*, which employs **Mutation-Based Fault Localisation (MBFL)** to localise the faults in DNN models introduced by wrong DL program configuration. Specifically, it mutates the faulty DNN models and implements two commonly used approaches in MBFL, Metallaxis and MUSE, and then filters out the most suspicious elements based on suspiciousness values of different mutants. These approaches are different from SpectAcle, as their assumption is that the fault is in the DL program; SpectAcle, instead, should be used to detect faults in the setting of the parameters of the model. In this sense, the approaches are complementary.

In comparison to all previous works of both groups, fault localisation in AI-enabled CPS faces new challenges, as stated in Section 1, including the lack of ground truth for DNN controllers, and the fact that a system execution involves a sequence of DNN controller inferences. These challenges make it impossible to apply the existing DNN fault localisation approaches in our context.

8.2 Fault Localisation of Classic CPSs

For classic CPS, there are several lines of work on their fault localisation. First, Liu et al. [32] present an iterative approach for localising faults in Simulink models by utilising decision trees; the approach uses decision trees to cluster similar failures and adopts two selection criteria to select the best cluster for fault localisation. Liu et al. [34] propose a fault localisation approach based on dynamic model slicing and statistical debugging; dynamic slicing identifies, for each test case, the blocks that are executed; then, classical suspiciousness metrics are used to rank the blocks in terms of suspiciousness. Liu et al. [33] propose a search algorithm to generate small and diverse test suites for improving fault localisation accuracy.

Second, Bartocci et al. [5] integrate STL monitoring into CPS fault localisation to refine the results. Later, they [6] develop CPSDebug to detect failures in Simulink models. Recently, Bartocci et al. [7] propose a search-based method to generate a passing test case that closely resembles the original failing test, which can extract precise information about the location of faults in the system.

All these works target CPS models (e.g., Simulink) that contain classic CPS blocks only, and so they are not applicable to AI-enabled CPSs. Indeed, the uninterpretable decision logic of DNNs makes it difficult to determine which DNN components are responsible for an inference. In our work, we propose using forward impact as a measurement for the influence of each neuron weights to the DNN inference.

9 Conclusion and Future Work

In this work, we propose the framework SpectAcle, that localises the DNN controller weights responsible for the system specification violation, by exploiting the sequence of DNN controller inferences. Our technical novelty consists in the construction of the execution spectrum, in which we take the system-level specification as the correctness criterion for the DNN controller, and use forward impact to calculate the relevance of each neuron weight to the DNN controller inference. Our experimental results show the effectiveness of SpectAcle, based on the comparison with two baseline approaches.

As a future work, we plan to devise other methods for computing forward impact, because this is the core factor that affects the effectiveness of the approach. For example, in this work, we consider the activation of a neuron as a possible cause of a wrong DNN control action. However, it could be that the non-activation of a neuron is responsible for a wrong action. As future work, we plan to devise a fault localisation approach in which also non-activation is considered as a possible cause of misbehaviour.

In SpectAcle, we consider all the inferences preceding the violation to do fault localisation. However, since it is more likely that a wrong control decision happens close to the violation, in SpectAcle$_{\text{W}}$ we use the Hamming window to weight more the inferences close to the violation. Since the tests considered in this work are not too long (i.e., they are just long enough to check whether they satisfy the specification), such a weighted approach is already very effective, as shown by our experiments. Still, the approach could benefit from a more detailed identification of the moment 'when' the control decision starts to be responsible for the misbehaviour. The theory of *actual causality* by Halpern and Pearl [21] requires three conditions to hold to claim that an event A is the cause of an event B: (**C1**) A must occur before B; (**C2**)if A does not happen, but all the other conditions of the state remain the same, then B does not happen; (**C3**) A is minimal, i.e., there is no subset of A that satisfies **C1** and **C2**. In our context, we consider A as the control action and B as the violation of the specification. It is easy to see that **C1** holds, as we check the time-points preceding the violation. However, guaranteeing condition **C2** is very difficult, as it requires to find, for each observed control action A in a test, another test in which only the control action A does not happen, but the remaining part of the state is the same. For an AI-enabled CPS, this would require to override the behaviour of the DNN controller and force a different control action to see if the violation does not occur. We leave as future work to design an approach that can do this in an effective manner.

References

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82, 11 (2009), 1780–1792. DOI: https://doi.org/10.1016/j.jss.2009.06.035

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. Van Gemund. 2006. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC '06)*. IEEE, 39–46. DOI: https://doi.org/10.1109/PRDC.2006.18

[3] J. Allen. 1977. Short term spectral analysis, synthesis, and modification by discrete Fourier transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 25, 3 (1977), 235–238. DOI: https://doi.org/10.1109/TASSP.1977.1162950

[4] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, 1–10. DOI: https://doi.org/10.1145/1985793.1985795

[5] Ezio Bartocci, Thomas Ferrère, Niveditha Manjunath, and Dejan Ničković. 2018. Localizing faults in simulink/stateflow models with STL. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (Part of CPS Week) (HSCC '18)*. ACM, New York, NY, 197–206. DOI: https://doi.org/10.1145/3178126.3178131

[6] Ezio Bartocci, Niveditha Manjunath, Leonardo Mariani, Cristinel Mateis, and Dejan Ničković. 2019. Automatic failure explanation in CPS models. In *Software Engineering and Formal Methods*. Peter Csaba Ölveczky and Gwen Salaün (Eds.), Springer International Publishing, Cham, 69–86.

[7] Ezio Bartocci, Leonardo Mariani, Dejan Ničković, and Drishti Yadav. 2022. Search-based testing for accurate fault localization in CPS. In *Proceedings of the 2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE '22)*, 145–156. DOI: https://doi.org/10.1109/ISSRE55969.2022.00024

[8] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. 2016. End to end learning for self-driving cars. arXiv:1604.07316. Retrieved from http://arxiv.org/abs/1604.07316

[9] Davide Li Calsi, Matias Duran, Xiao-Yi Zhang, Paolo Arcaini, and Fuyuki Ishikawa. 2023. Distributed repair of deep neural networks. In *Proceedings of the 2023 IEEE Conference on Software Testing, Verification and Validation (ICST '23)*. IEEE, IEEE Computer Society, Los Alamitos, CA, 83–94.

[10] Jialun Cao, Meiziniu Li, Xiao Chen, Ming Wen, Yongqiang Tian, Bo Wu, and Shing-Chi Cheung. 2022. DeepFD: Automated fault diagnosis and localization for deep learning programs. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. ACM, New York, NY, 573–585. DOI: https://doi.org/10.1145/3510003.3510099

[11] Jacob Cohen. 1969. *Statistical Power Analysis for the Behavioral Sciences*. Academic Press, New York.

[12] Alexandre Donzé and Oded Maler. 2010. Robust satisfaction of temporal logic over real-valued signals. In *Proceedings of the 8th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS '10)*. Springer-Verlag, Berlin, 92–106.

[13] Tommaso Dreossi, Alexandre Donzé, and Sanjit A. Seshia. 2019. Compositional falsification of cyber-physical systems with machine learning components. *Journal of Automated Reasoning* 63, 4 (Dec. 2019), 1031–1053. DOI: https://doi.org/10.1007/s10817-018-09509-5

[14] Matias Duran, Xiao-Yi Zhang, Paolo Arcaini, and Fuyuki Ishikawa. 2021. What to blame? On the granularity of fault localization for deep neural networks. In *Proceedings of the 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE '21)*, 264–275. DOI: https://doi.org/10.1109/ISSRE52982.2021.00037

[15] Hasan Ferit Eniser, Simos Gerasimou, and Alper Sen. 2019. DeepFault: Fault localization for deep neural networks. In *Fundamental Approaches to Software Engineering*. Reiner Hähnle and Wil van der Aalst (Eds.), Springer International Publishing, Cham, 171–191.

[16] Gidon Ernst, Paolo Arcaini, Georgios Fainekos, Federico Formica, Jun Inoue, Tanmay Khandait, Mohammad Mahdi Mahboob, Claudio Menghi, Giulia Pedrielli, Masaki Waga, et al. 2022. ARCH-COMP 2022 category report: Falsification with ubounded resources. In *Proceedings of 9th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH '22)*. Goran Frehse, Matthias Althoff, Erwin Schoitsch, and Jeremie Guiochet (Eds.), EPiC Series in Computing, Vol. 90, EasyChair, 204–221. DOI: https://doi.org/10.29007/fhnk

[17] Georgios E. Fainekos and George J. Pappas. 2009. Robustness of temporal logic specifications for continuous-time signals. *Theoretical Computer Science* 410, 42 (Sep. 2009), 4262–4291. DOI: https://doi.org/10.1016/j.tcs.2009.06.021

[18] Ali Ghanbari, Deepak-George Thomas, Muhammad Arbab Arshad, and Hridesh Rajan. 2023. Mutation-based fault localization of deep neural networks. In *Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE '23)*, 1301–1313. DOI: https://doi.org/10.1109/ASE56229.2023.00171

[19] Philip E. Gill, Walter Murray, and Margaret H. Wright. 2019. *Practical Optimization*. SIAM.

[20] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. 2017. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *Proceedings of the 2017 IEEE International Conference on Robotics and Automation (ICRA '17)*. IEEE Press, 3389–3396. DOI: https://doi.org/10.1109/ICRA.2017.7989385

[21] Joseph Y. Halpern. 2016. *Actual Causality*. The MIT Press.

[22] Chao Huang, Jiameng Fan, Wenchao Li, Xin Chen, and Qi Zhu. 2019. ReachNN: Reachability analysis of neural-network controlled systems. *ACM Transactions on Embedded Computing Systems* 18, 5s, Article 106 (Oct. 2019), 22 pages. DOI: https://doi.org/10.1145/3358228

[23] Xiaoqing Jin, Jyotirmoy V. Deshmukh, James Kapinski, Koichi Ueda, and Ken Butts. 2014. Powertrain control verification benchmark. In *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control (HSCC '14)*. ACM, New York, NY, 253–262. DOI: https://doi.org/10.1145/2562059.2562140

[24] Taylor T. Johnson, Diego Manzanas Lopez, Luis Benet, Marcelo Forets, Sebastián Guadalupe, Christian Schilling, Radoslav Ivanov, Taylor J. Carpenter, James Weimer, and Insup Lee. 2021. ARCH-COMP21 category report: Artificial Intelligence and Neural Network Control Systems (AINNCS) for continuous and hybrid systems plants. In *Proceedings*

of the 8th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH '21). Goran Frehse and Matthias Althoff (Eds.), EPiC Series in Computing, Vol. 80, EasyChair, 90–119. DOI: https://doi.org/10.29007/kfk9

[25] Taylor T. Johnson, Diego Manzanas Lopez, Patrick Musau, Hoang-Dung Tran, Elena Botoeva, Francesco Leofante, Amir Maleki, Chelsea Sidrane, Jiameng Fan, and Chao Huang. 2020. ARCH-COMP20 category report: Artificial Intelligence and Neural Network Control Systems (AINNCS) for continuous and hybrid systems plants. In *Proceedings of the 7th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH '20)*. Goran Frehse and Matthias Althoff (Eds.), EPiC Series in Computing, Vol. 74, EasyChair, 107–139. DOI: https://doi.org/10.29007/9xgv

[26] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. ACM, New York, NY, 467–477. DOI: https://doi.org/10.1145/581339.581397

[27] Tanmay Khandait, Federico Formica, Paolo Arcaini, Surdeep Chotaliya, Georgios Fainekos, Abdelrahman Hekal, Atanu Kundu, Ethan Lew, Michele Loreti, Claudio Menghi, et al. 2024. ARCH-COMP 2024 category report: Falsification. In *Proceedings of the 11th International Workshop on Applied Verification for Continuous and Hybrid Systems*. Goran Frehse and Matthias Althoff (Eds.), EPiC Series in Computing, Vol. 103, EasyChair, 122–144. DOI: https://doi.org/10.29007/hgfv

[28] Barbara Kitchenham, Lech Madeyski, David Budgen, Jacky Keung, Pearl Brereton, Stuart Charters, Shirley Gibbs, and Amnart Pohthong. 2017. Robust statistical methods for empirical software engineering. *Empirical Software Engineering* 22 (2017), 579–630.

[29] Tien-Duy B. Le, Ferdian Thung, and David Lo. 2013. Theory and practice, do they match? A case with spectrum-based fault localization. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM '13)*. IEEE Computer Society, 380–383. DOI: https://doi.org/10.1109/ICSM.2013.52

[30] Davide Li Calsi, Matias Duran, Thomas Laurent, Xiao-Yi Zhang, Paolo Arcaini, and Fuyuki Ishikawa. 2023. Adaptive search-based repair of deep neural networks. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '23)*. ACM, New York, NY, 1527–1536. DOI: https://doi.org/10.1145/3583131.3590477

[31] Tsang-Wei Lin, Sheue-Ling Hwang, and Paul A. Green. 2009. Effects of time-gap settings of adaptive cruise control (ACC) on driving performance and subjective acceptance in a bus driving simulator. *Safety Science* 47, 5 (2009), 620–625. DOI: https://doi.org/10.1016/j.ssci.2008.08.004

[32] Bing Liu, Lucia, Shiva Nejati, Lionel Briand, and Thomas Bruckmann. 2016. Localizing multiple faults in simulink models. In *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER '16)*, Vol. 1, 146–156. DOI: https://doi.org/10.1109/SANER.2016.38

[33] Bing Liu, Lucia, Shiva Nejati, and Lionel C. Briand. 2017. Improving fault localization for Simulink models using search-based testing and prediction models. In *Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER '17)*, 359–370. DOI: https://doi.org/10.1109/SANER.2017.7884636

[34] Bing Liu, Lucia, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. 2016. Simulink fault localization: An iterative statistical debugging approach. *Software Testing, Verification and Reliability* 26, 6 (2016), 431–459.

[35] Chen Lv, Yang Xing, Junzhi Zhang, Xiaoxiang Na, Yutong Li, Teng Liu, Dongpu Cao, and Fei-Yue Wang. 2018. Levenberg–Marquardt backpropagation training of multilayer neural networks for state estimation of a safety-critical cyber-physical system. *IEEE Transactions on Industrial Informatics* 14, 8 (2018), 3436–3446. DOI: https://doi.org/10.1109/TII.2017.2777460

[36] Deyun Lyu, Zhenya Zhang, Paolo Arcaini, Fuyuki Ishikawa, Thomas Laurent, and Jianjun Zhao. 2024. Search-based repair of DNN controllers of AI-enabled cyber-physical systems guided by system-level specifications. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '24)*. ACM, New York, NY, 1435–1444. DOI: https://doi.org/10.1145/3638529.3654078

[37] Deyun Lyu, Zhenya Zhang, Paolo Arcaini, Xiao-Yi Zhang, Fuyuki Ishikawa, and Jianjun Zhao. 2024. Code and benchmarks of the paper "SpectAcle: Fault localisation of AI-enabled CPS by exploiting sequences of DNN controller inferences". Retrieved from https://github.com/lyudeyun/Spectacle

[38] Deyun Lyu, Zhenya Zhang, Paolo Arcaini, Xiao-Yi Zhang, Fuyuki Ishikawa, and Jianjun Zhao. 2024. Supplementary material for the paper "SpectAcle: Fault localisation of AI-enabled CPS by exploiting sequences of DNN controller inferences". Retrieved from https://sites.google.com/view/spectacle4aicps

[39] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: Automated neural network model debugging via state differential analysis and input selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 175–186.

[40] Diego Manzanas Lopez, Matthias Althoff, Luis Benet, Xin Chen, Jiameng Fan, Marcelo Forets, Chao Huang, Taylor T. Johnson, Tobias Ladner, Wenchao Li, et al. 2022. ARCH-COMP22 category report: Artificial Intelligence and Neural Network Control Systems (AINNCS) for continuous and hybrid systems plants. In *Proceedings of 9th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH '22)*. Goran Frehse, Matthias Althoff,

Erwin Schoitsch, and Jeremie Guiochet (Eds.), EPiC Series in Computing, Vol. 90, EasyChair, 142–184. DOI: https://doi.org/10.29007/wfgr

[41] Diego Manzanas Lopez, Matthias Althoff, Marcelo Forets, Taylor T. Johnson, Tobias Ladner, and Christian Schilling. 2023. ARCH-COMP23 category report: Artificial Intelligence and Neural Network Control Systems (AINNCS) for continuous and hybrid systems plants. In *Proceedings of 10th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH '23)*. Goran Frehse and Matthias Althoff (Eds.), EPiC Series in Computing, Vol. 96, EasyChair, 89–125. DOI: https://doi.org/10.29007/x38n

[42] Mathworks. 2023. Simulink. Retrieved from https://www.mathworks.com/products/simulink.html

[43] Claudio Menghi, Paolo Arcaini, Walstan Baptista, Gidon Ernst, Georgios Fainekos, Federico Formica, Sauvik Gon, Tanmay Khandait, Atanu Kundu, Giulia Pedrielli, et al. 2023. ARCH-COMP23 category report: Falsification. In *Proceedings of 10th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH '23)*. Goran Frehse and Matthias Althoff (Eds.), EPiC Series in Computing, Vol. 96, EasyChair, 151–169. DOI: https://doi.org/10.29007/6nqs

[44] Martin Fodslette Møller. 1993. A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks* 6, 4 (1993), 525–533.

[45] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology* 20, 3, Article 11 (Aug. 2011), 32 pages. DOI: https://doi.org/10.1145/2000791.2000795

[46] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE '17)*. IEEE, 609–620.

[47] Jeongju Sohn, Sungmin Kang, and Shin Yoo. 2023. Arachne: Search-based repair of deep neural networks. *ACM Transactions on Software Engineering and Methodology* 32, 4, Article 85 (May 2023), 26 pages. DOI: https://doi.org/10.1145/3563210

[48] Jiayang Song, Deyun Lyu, Zhenya Zhang, Zhijie Wang, Tianyi Zhang, and Lei Ma. 2022. When cyber-physical systems meet AI: A benchmark, an evaluation, and a way forward. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22)*. ACM, New York, NY, 343–352. DOI: https://doi.org/10.1145/3510457.3513049

[49] Shogo Tokui, Susumu Tokumoto, Akihito Yoshii, Fuyuki Ishikawa, Takao Nakagawa, Kazuki Munakata, and Shinji Kikuchi. 2022. NeuRecover: Regression-controlled repair of deep neural networks with training history. In *Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '22)*. IEEE Computer Society, Los Alamitos, CA, 1111–1121. DOI: https://doi.org/10.1109/SANER53432.2022.00128

[50] Hoang-Dung Tran, Feiyang Cai, Manzanas Lopez Diego, Patrick Musau, Taylor T. Johnson, and Xenofon Koutsoukos. 2019. Safety verification of cyber-physical systems with reinforcement learning control. *ACM Transactions on Embedded Computing Systems* 18, 5s, Article 105 (Oct. 2019), 22 pages. DOI: https://doi.org/10.1145/3358230

[51] Hoang-Dung Tran, Xiaodong Yang, Diego Manzanas Lopez, Patrick Musau, Luan Viet Nguyen, Weiming Xiang, Stanley Bak, and Taylor T. Johnson. 2020. NNV: The Neural Network Verification tool for deep neural networks and learning-enabled cyber-physical systems. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 3–17.

[52] Mohammad Wardat, Breno Dantas Cruz, Wei Le, and Hridesh Rajan. 2022. DeepDiagnosis: Automatically diagnosing faults and recommending actionable fixes in deep learning programs. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. ACM, New York, NY, 561–572. DOI: https://doi.org/10.1145/3510003.3510071

[53] Mohammad Wardat, Wei Le, and Hridesh Rajan. 2021. DeepLocalize: Fault localization for deep neural networks. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*. IEEE Press, 251–262. DOI: https://doi.org/10.1109/ICSE43902.2021.00034

[54] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. 2012. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.

[55] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2013. The DStar method for effective software fault localization. *IEEE Transactions on Reliability* 63, 1 (2013), 290–308.

[56] W. Eric Wong, Vidroha Debroy, Yihao Li, and Ruizhi Gao. 2012. Software fault localization using DStar (D*). In *Proceedings of the 2012 IEEE 6th International Conference on Software Security and Reliability (SERE '12)*. IEEE Computer Society, 21–30. DOI: https://doi.org/10.1109/SERE.2012.12

[57] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (Aug. 2016), 707–740. DOI: https://doi.org/10.1109/TSE.2016.2521368

[58] Shakiba Yaghoubi and Georgios Fainekos. 2019. Gray-box adversarial testing for control systems with machine learning components. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control (HSCC '19)*. ACM, New York, NY, 179–184. DOI: https://doi.org/10.1145/3302504.3311814

[59] Xiao-Yi Zhang and Mingyue Jiang. 2021. SPICA: A methodology for reviewing and analysing fault localisation techniques. In *Proceedings of the 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME '21)*. IEEE, 366–377. DOI: https://doi.org/10.1109/ICSME52107.2021.00039

[60] Zhenya Zhang, Deyun Lyu, Paolo Arcaini, Lei Ma, Ichiro Hasuo, and Jianjun Zhao. 2023. FalsifAI: Falsification of AI-enabled hybrid control systems guided by time-aware coverage criteria. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1842–1859. DOI: https://doi.org/10.1109/TSE.2022.3194640

Received 1 March 2024; revised 7 August 2024; accepted 2 November 2024