This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3200421

Self-Checking Deep Neural Networks for Anomalies and Adversaries in Deployment

Yan Xiao, Ivan Beschastnikh, Yun Lin, Rajdeep Singh Hundal,
Xiaofei Xie, David S. Rosenblum, Jin Song Dong

Abstract—Deep Neural Networks (DNNs) have been widely adopted, yet DNN models are surprisingly unreliable, which raises significant concerns about their use in critical domains. In this work, we propose that runtime DNN mistakes can be quickly detected and properly dealt with *in deployment*, especially in settings like self-driving vehicles. Just as software engineering (SE) community has developed effective mechanisms and techniques to monitor and check programmed components, our previous work, SelfChecker, is designed to monitor and correct DNN predictions given unintended abnormal test data. SelfChecker triggers an alarm if the decisions given by the internal layer features of the model are inconsistent with the final prediction and provides *advice* in the form of an alternative prediction. In this paper, we extend SelfChecker to the security domain. Specifically, we describe SelfChecker++, which we designed to target both *unintended* abnormal test data and *intended* adversarial samples. Technically, we develop a technique which can transform any runtime inputs triggering alarms into semantically equivalent inputs, then we feed those transformed inputs to the model. Such runtime transformation nullifies any intended crafted samples, making the model immune to adversarial attacks that craft adversarial samples. We evaluated the alarm accuracy of SelfChecker++ on three DNN models and four popular image datasets, and found that SelfChecker++ triggers correct alarms on 63.10% of wrong DNN predictions, and triggers false alarms on 5.77% of correct DNN predictions. We also evaluated the effectiveness of SelfChecker++ in detecting adversarial examples and found it detects on average 70.09% of such samples with advice accuracy that is 20.89% higher than the original DNN model and 18.37% higher than SelfChecker.

Index Terms—self-checking system, trustworthiness, deep neural networks, adversarial examples, deployment

1 Introduction

Deep Neural Networks (DNNs) have achieved high performance across many domains, such as speech processing [1], image classification [2], medical diagnostics [3], and social media [4]. DNNs have been deployed to support many mission-critical applications, including malware detection [5], aircraft detection systems [6] and autonomous vehicles [7]. There are increasing concerns about the *generality* of deep learning models, i.e., their limited performance on unseen samples beyond the training dataset. For example, such models may behave in unexpected ways when processing either unintended anomalies or adversarial inputs. In safety- and security-critical applications an incorrect DNN decision could be costly. We believe that such applications must include deployment-time logic to:

1) *check* the trustworthiness of a DNN's prediction,
2) raise an *alarm* when there is low confidence in the output, e.g., when encountering adversarial examples, and
3) provide an alternative prediction that we term *advice*, just as software engineering (SE) community has developed such methods for programmed components [8], [9].

Trustworthiness of simple DNNs can be estimated with mutual information [10] and softmax probabilities [11]. However, softmax probabilities are unreliable confidence estimators of the prediction in complex DNNs with many layers and neurons [12], [13] since DNNs may still produce overconfident posterior probabilities even for abnormal samples. To address unexpected runtime samples, intended or unintended, existing state-of-the-art can be classified into three categories.

Auxiliary Model Construction. Several prior projects considered the problem of evaluating the trustworthiness of a deployed DNN by introducing new auxiliary models. Stocco et al. [14] built a collection of autoencoder models to check the confidence of a decision from a self-driving car model. Their approach, however, is specific to self-driving car models whose inputs are image frame sequences from videos that change little over time. DISSECTOR, proposed by Wang et al. [15] validates inputs that represent deviations from normal inputs. This work does not describe how to set the deviation threshold, which will vary for different models and datasets. Several projects in the deep learning (DL) community [13], [16], [17] developed new learning-based models to measure the confidence of original pre-trained DL models. ConfidNet, proposed by Corbière et al. [13] is a new confidence model learning the confidence criterion for failure prediction built on top of the pre-trained model. ConfidNet models may also be untrustworthy and may suffer from overfitting. However, none of above techniques contend with an input instance that triggers an alarm. Only in [14] is human interference introduced to consider the alarm. We propose that, though human involvement is ideal, humans are not always available to provide input. Instead, we believe that automatically generated advice is an important and necessary way of coping with runtime

- *Yan Xiao, Yun Lin, Rajdeep Singh Hundal, and Jin Song Dong are with School of Computing, National University of Singapore, Singapore.*
- *Ivan Beschastnikh is with Department of Computer Science, University of British Columbia, Vancouver, BC, Canada.*
- *Xiaofei Xie is with School of Computing and Information Systems, Singapore Management University, Singapore.*
- *David S. Rosenblum is with Department of Computer Science, George Mason University, Fairfax, VA, USA.*

alarms triggered by deviating inputs.

Model Training. Data augmentation [18], [19] and adversarial training [20], [21] are popular techniques. However, it is impossible to include all kinds of adversarial examples with data augmentation and all possible perturbations with adversarial training. Therefore, data augmentation cannot defend against unseen adversarial examples. And both techniques significantly increase the training cost.

Anomaly Assumption. Wang et al. [22] proposed mMutant to distinguish adversarial from normal inputs. They make an assumption that adversarial samples usually lie around classification boundaries. This work was superseded by DISSECTOR which also has a threshold assumption of its own. In our work we observed that the assumptions supporting these techniques fail to hold in practice (see Section 5.3 for more details). Besides, neither mMutant nor DISSECTOR provide advice once an adversarial example is detected.

Our goal is to build a general-purpose system that (1) checks a deployed DNN's predictions in deployment, (2) raises an *alarm* if there are any anomalies, and (3) provides an alternative prediction that we term *advice*. The first key challenge in building such a system is finding a source of additional information to check DNN outputs. The second challenge is how to make the system immune to attacks while preserving model prediction accuracy.

In our previous work we developed SelfChecker [23], which works only on unintended samples with limited distribution shift from the training dataset. In this paper we propose a novel self-checking system, **SelfChecker++**, to address the challenges raised by both unintended anomalies and intentional adversaries. Similar to SelfChecker, SelfChecker++ (1) triggers an alarm if the classes inferred by most internal layer features of the model are inconsistent with the final prediction, and (2) provides *advice* in the form of an alternative prediction. However, unlike SelfChecker, SelfChecker++ (1) relaxes the assumption of SelfChecker that the training and validation datasets come from a distribution similar to that of the inputs that the DNN model will face in deployment, and (2) makes the model immune to adversarial attacks in which an adversary crafts adversarial inputs. We designed SelfChecker++ by introducing a GAN-based (Generative Adversarial Network) transformation technique to learn the distribution of the training data to differentiate samples from a different distribution, and synthesize a sample from the latent space (avoiding the processing of any potential adversarial samples).

To evaluate SelfChecker++'s alarm and advice mechanisms in the image classification domain, we conducted experiments to check three neural networks on four datasets. The networks in particular were ConvNet, VGG-16, and ResNet-20. The datasets on the other hand were MNIST, FMNIST, CIFAR-10, and CIFAR-100. Moreover, these datasets are widely used and available to the public. Furthermore, we compared SelfChecker++ against three existing approaches (SELFORACLE [14], ConfidNet [13], and DISSECTOR [15]). From the results we attained, SelfChecker++ has an F1-score of 67.92%. This is also the best score, 9.15% more than the runner up, ConfidNet. With respect to self-driving cars, SelfChecker++ was also put to the test against state-of-the-art techniques like SELFORACLE [14]. It was then observed that SelfChecker++ raised more right alarms and were comparable in wrong alarms. Lastly, our evaluation of detecting adversarial examples, SelfChecker++ achieves 34.54% higher F1-score than DISSECTOR. For the advice accuracy, SelfChecker++ (average 69.57%) performs much better than SelfChecker (average 51.20%).

Differences from SelfChecker. This work extends our previous work on SelfChecker [23] in the following ways:

1) We introduce a new technique that uses a GAN to provide more accurate advice. To learn additional information from the training dataset, we train a GAN to synthesize inputs which conform to the distribution of the training dataset given random latent vectors. For input instances that trigger an alarm, we use this GAN to generate alternative inputs that are semantically-preserving and conform to the training data distribution (Section 4.3). We add corresponding background and motivating examples to Section 3.
2) We discuss the threat model for our work and our objectives (Section 2).
3) We add the alarm accuracies of checking ResNet-20 and ConvNet on MNIST and CIFAR-100 (Section 5.3.1).
4) We empirically present SelfChecker++'s effectiveness in detecting adversarial examples and explain how it differs from existing work (Section 5.3.2), report new advice accuracies on not only normal test data but also adversarial examples (Section 5.3.3), and discuss the necessity of alarm analysis (Section 5.3.4).
5) We survey studies related to adversarial examples (Section 6) and discuss threats to validity of our work (Section 7).

In summary, our paper makes the following three contributions:

⋆ We present the design of SelfChecker++, which uses density distributions of layer features and a search-based layer selection strategy to trigger an alarm if a DNN model output has low confidence. We show that SelfChecker++ achieves better alarm accuracy than previous work on both unintended anomalies and adversarial inputs.
⋆ Unlike existing work, SelfChecker++ provides *advice* in the form of an alternative prediction. This prediction is generated from a semantically-preserving input that is synthesized by a GAN model.
⋆ We demonstrate the effectiveness of SelfChecker++'s alarms and advice on publicly available DNNs, ranging from small models (ConvNet) to large and complex models (VGG-16 and ResNet-20), and self-driving car scenarios. Our implementation is open-source[1].

2 THREAT MODEL

We consider the following threat model in this work. Our work focuses on victim DNN models in the image classification domain. Let f be a learned function by a DNN model M, which is an image classifier. Given an image input instance x whose DNN's output is y, i.e., $y = f(x)$ an attacker can intentionally or unintentionally make the model fail in two ways.

1. https://github.com/yanxiao6/SelfCheckerPlusPlus

Intentional Attack. The attacker can design an adversarial example ($x + \delta$) to fool M into making a wrong decision y' (i.e., $y' = f(x + \delta)$) where $y' \neq y$. However, to a human observer [24], [25], [26], the image ($x + \delta$) will not semantically differ from x.

Unintentional Attack. The attacker can unintentionally send an input $\hat{x}$, which is sampled from a different distribution than the one used for training M. From a human perspective [24], [25], [26], $\hat{x}$ has a semantical label y while $y' = f(\hat{x})$ and $y' \neq y$. In other words, such a distribution shift causes M to fail on the sample $\hat{x}$.

Our first goal is to detect such adversarial examples and trigger alarms. Our second goal is to synthesize images that are similar to the original ones ($x + \delta$ and $\hat{x}$) but conform to the distribution of the training data. We use these generated images to provide an alternative prediction (we call this an *advice*) as a defense mechanism.

3 Background and Motivation

3.1 Deep Learning

Deep learning comprises of neural networks that are built with three basic layers. Firstly, we have the input layer and it is, as it is so adequately named, where the input goes. Secondly, we have the hidden layer and it is where activation functions on top of neurons extract the desired features from the given input. The third and final layer is called the output layer which tries to generate a prediction for the given problem. Moreover, there are many approaches used to generate the prediction, with the common ones being *classification* and *regression*. Classification utilizes categorical classes whereas regression utilizes real-valued ordinals. Each layer except the input layer, extracts more abstract features than the preceding layer.

A DNN can be thought of as a mapping function F. Given an input vector $\mathbf{x}$, it is then mapped to an output vector in a cascading system of non-linear activation functions, $f_{1...L}$, of L hidden layers and corresponding weight parameters ($\mathbf{W}_{1...L}$ and $b_{1...L}$)

$$F(\mathbf{x}) = f_1(\mathbf{W}_1 f_2(...\mathbf{W}_{L-1} f_L(\mathbf{W}_L \mathbf{x} + b_L) + b_{L\text{-}1}...) + b_1) \quad (1)$$

The weights for a typical network are discovered during the training phase (i.e., with training data) using an iterative training process, such as stochastic gradient descent in conjunction with backward propagation [27]. Internally, the computation of the decision function involves each neuron in one layer computing its activation function over the outputs of the previous layer. A layer's activation outputs describe the exact behavior of that particular layer during the inference phase. In this paper, *layer features* are with reference to the vectors of activation outputs from the layers in a network.

3.2 Generative Adversarial Network (GAN)

Goodfellow et al. [28] proposed GANs to synthesize realistic data. A GAN contains two neural networks, a generative net G to generate images and a discriminator net D to classify the generated images as either real or fake. Both models are trained adversarially in order to continually improve images generated by G that can fool D while D tries to differentiate the generated images from the real ones. In the training phase, random vectors z (assumed to be isotropic Gaussian) and training data samples x are fed into G so that the generated samples $G(z)$ would attain a distribution that is almost identical to x. The discriminator D then learns to distinguish $G(z)$ from x (i.e., fake from real samples). To correctly classify $G(z)$ as fake and x as real, the GAN is trained to minimize a loss function:

$$\begin{aligned} min_G max_D V(D,G) = & E_{x \sim p_r(x)}[log(D(x))] \\ & + E_{z \sim p_g(z)}[log(1 - D(G(z)))] \end{aligned} \quad (2)$$

where p_r and p_g are the real and generated distributions respectively. Essentially, the training goal here is to make p_g close to p_r.

However, in general, it can be hard to strike a balance between G and D during training. When D becomes too good too fast, G fails to learn efficiently and its loss function saturates. To mitigate this, the original authors proposed to train G to instead maximize $logD(G(z))$. This is known as the non-saturating loss.

DCGAN proposed by Radford et al. [29], is an extension which uses deep convolutional networks instead of multilayer perceptrons. It is by far the most commonly used GAN variation as it tends to be more stable in practice.

Lastly, when it comes to the evaluation of GANs, there are two metrics commonly used in practice. Salimans et al. [30] proposed the Inception Score (IS) which essentially utilizes a pre-trained network to classify a substantial amount of generated images. Heusel et al. [31] later suggested the Fréchet Inception Distance (FID) which utilizes a pre-trained network along with a substantial amount of generated and real images. By incorporating real images, the FID metric more accurately captures the similarity between generated and real images.

3.3 The Promise of Using Layer Features

A neural network fundamentally uses the features it has learnt from training to make the appropriate decisions during testing. The question then has to be asked as to how is it possible to determine if a model decides incorrectly for any given instance $\mathbf{x}$ during testing? A simple approach is to determine if a similar training instance $\mathbf{x}'$ has been seen by the model before. However, how would one then measure the similarity between $\mathbf{x}$ and $\mathbf{x}'$? A majority of the current methods utilize a distance-based measure [32]. Some examples of this are L_p and cosine similarity. In contrast, we think that this approach is not optimal because the inputs are already incredibly complex to the point where DNNs are employed to learn features from it. Therefore, it is doubtful that input similarity can be adequately represented from just a distance measure itself.

Our solution to capturing similarity is to utilize DNN internal layer features. In particular, similarity is now the probability that a DNN saw similar layer features beforehand (i.e., whilst training). Using probability density distributions, we are able to measure the similarity between $\mathbf{x}$ and $\mathbf{x}'$. These distributions were extrapolated from the training phase.

Fig. 1 depicts a motivating scenario, a typical Convolutional Neural Network (CNN) whose architecture consists

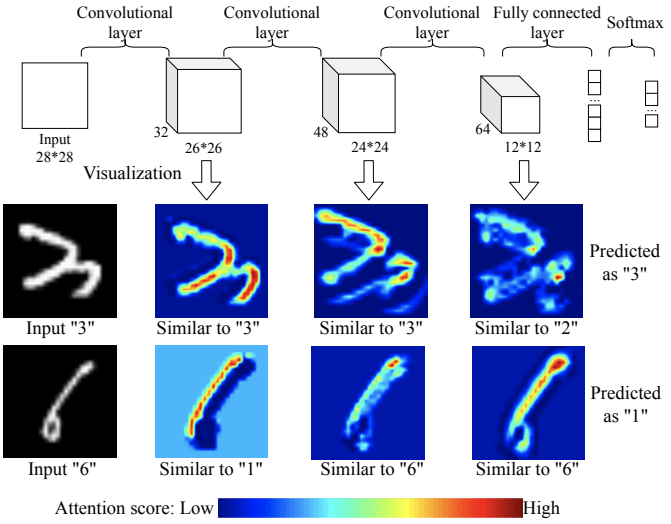


Fig. 1. The topmost portion of the figure depicts a typical trained Convolutional Neural Network whilst the bottom two rows shows the attention heatmaps for each convolutional layer with respect to the input images.

of three convolutional layers followed by a fully connected and softmax layer. The dataset used in Fig. 1 is MNIST (i.e., digit images) and in short, the model attempts to classify these images. In order to further aid our understanding on the difference between each layer's features, we employ Grad-CAM [33] to picture the attention each layer has with respect to the input via heatmaps. Looking closely at the heatmaps, it can be observed that the area of focus varies from layer to layer. More specifically, considering the first row (i.e., digit 3), we can observe that the last image resembles digit 2 instead. For the second row (i.e., digit 6), we can observe that the first image resembles digit 1 instead.

From Fig. 1 we can see that, digit 6 is incorrectly predicted as digit 1. However, most of the hidden layers disagree with the finial prediction and recognize the digit as 1 correctly. Similarly, most of the hidden layers recognize digit 3 correctly as well. Thus, this shows that using layer features to further scrutinize or confirm a model's prediction is indeed applicable.

There are multiple variations of DNNs and they can also be merged together to increase the overall complexity of the networks. One such scenario where this happens is urban flow prediction [34]. Networks here are from a combination of convolutional, graph and recurrent networks. It should be noted that all these networks utilize internal layers in order to learn useful features, which is the basis of our research.

The technique we detail is more tailored towards classification networks which include convolutional and fully-connected layers. That being said, it is also applicable towards regression networks by simply modifying the regression into binary classification. Furthermore, because our technique utilizes layer features, it is bound to work on other network variations as well (e.g., recurrent networks). However, we evaluate our technique on other network variations in future work.

3.4 The Challenges of Using Layer Features

Fig. 1 also brings about some challenges to address when utilizing layer features:

- Which layers are to be used when checking if a prediction is correct? For instance, does the amount of layers used matter?
- Feature aggregation from multiple layers — how should it be done when triggering an alarm and providing advice?

Answering the aforementioned questions to optimize alarm and advice accuracy on a single dataset and a specific DNN architecture is sufficiently difficult. We go a step further and design SelfChecker++ to both yield high performance and also generalize across datasets and DNN architectures (different number of layers, neurons in a layer). The generality of our approach is the key contribution of our paper.

Problem statement. Our goal is to come up with a systematic method named SelfChecker++ which would be used for detecting if a trained DNN incorrectly classified a test instance. SelfChecker++ accomplishes this by doing a thorough check on the network's internal features. SelfChecker++ first is tasked with raising an *alarm* if it deems that an instance was incorrectly classified. Following the *alarm* and going beyond existing works [13], [14], [15], SelfChecker++ is then tasked with giving *advice* (i.e., an alternative prediction). SelfChecker++ should also attain high accuracies when it comes to raising alarms and giving advice.

3.5 Using a GAN to synthesize improved inputs

An input instance in deployment may contain noise perturbations or large distribution shifts (e.g., blurry images). Even though the deployed DNN model may be well-trained, it may still make wrong predictions on such inputs. However, if these instances can be transformed by preserving their semantic information into inputs that better conform to the training data distribution, the DNN model can make better predictions on these transformed instances. This is the key idea behind our use of a GAN in this work.

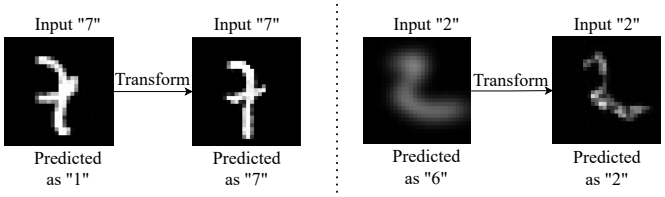


Fig. 2. Examples of improved inputs generated by a GAN.

Figure 2 presents two motivating examples that are generated by a GAN we designed for input images of digits "7" and "2". The original input image of a "7" has a crooked middle horizontal line and a curved up line leading to the wrong prediction of "1". The GAN-generated transformation, however, is correctly predicted by the model as "7" since the middle line became straighter and the edge between the top line and vertical line became sharper. The right column shows a blurry "2" image. The DNN model wrongly classifies this image as a "6", but correctly classifies the rightmost, GAN transformed, image as a "2". These two examples motivate the power of the GAN-based input transformation process.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3200421

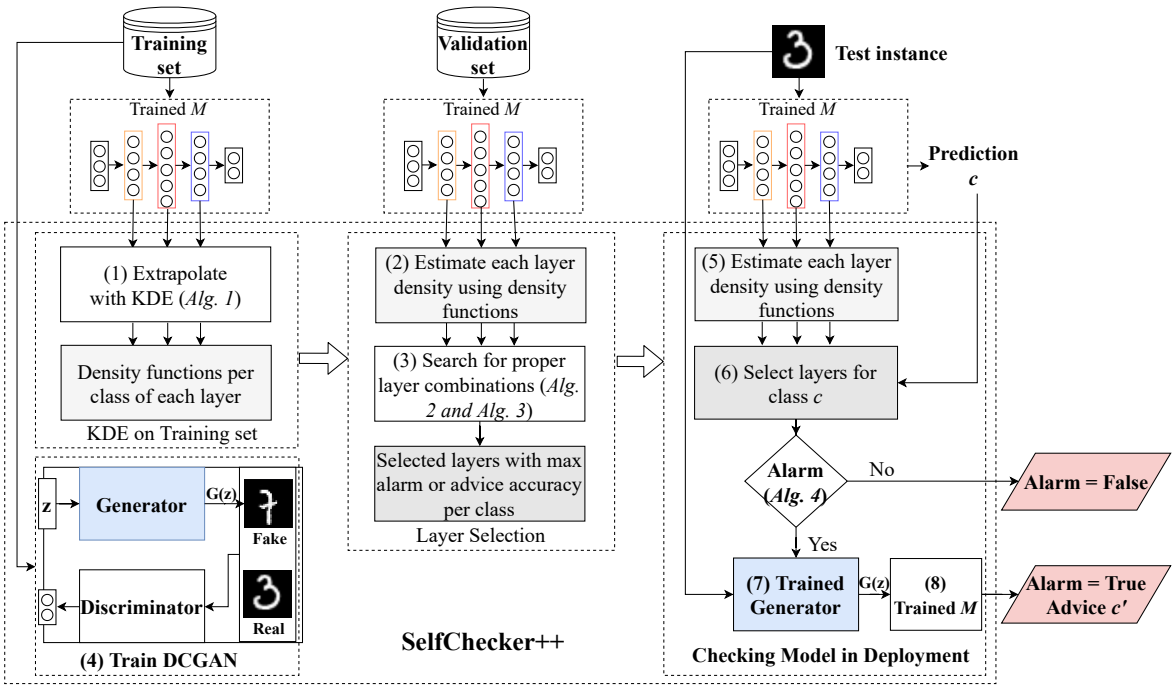


Fig. 3. The design of SelfChecker++ and its integration with a trained model and model predictions. The bold parts are the main differences between SelfChecker and SelfChecker++.

4 Design of SelfChecker++

SelfChecker++ has three primary goals — to analyze a DNN's prediction, trigger an *alarm* should the prediction be deemed wrong and give *advice* following the alarm.

SelfChecker++ has two modules. The *training module* is used in conjunction with the already trained DNN model and utilizes both training and validation datasets in order to set the necessary configurations for deployment. These configurations would directly affect how SelfChecker++ behaves. The DCGAN is also trained in the training module.

The *deployment module* on the other hand is utilized in conjunction with the inference process. In particular, it checks the DNN's internal features with respect to a test instance and raises an alarm if the DNN's prediction is deemed to be inconsistent. It accomplishes this with the aid of the configuration from the *training module*. The trained DCGAN is used to provide the advice for the input instances that raise alarms.

Even though SelfChecker++ checks the internal features of a DNN, the network need not undergo any modifications or retraining. The prime reason for this stems from the fact that the training module is independent from DNN architectures. In contrast, the deployment module is DNN specific.

The framework of SelfChecker++ is depicted in Fig. 3. Its initial basis is the network M that is trained and validated using datasets D_{train} and D_{valid} respectively. Firstly, the training module utilizes kernel density estimation (KDE) [35] to (1) calculate the density distributions for each class at every layer with respect to D_{train} (see Section 4.1 for more information). Subsequently, with the aid of the distributions in (1), (2) SelfChecker++ is now able to, for every class, approximate the density values for any instance (i.e., either from the validation or test dataset). The key idea here is that, given an instance, large density values for a specific class at a layer would indicate that the features of the current instance at that layer are similar to that of the class. Once SelfChecker++ attains the density values of the entire validation dataset (for every layer), it then (3) searches for the optimal layer combinations. Optimality here is with reference to attaining the best accuracies for both alarm and advice. Moreover, global search is utilized to locate these layer combinations for each class (see Section 4.2 for more information). This is because, the feature behaviors for different classes at different layers vary and are quite unique. To provide advice for those input triggering alarms, (4) DCGAN is trained on D_{train}.

Lastly, during deployment, the deployment module determines if it should raise an alarm and subsequently give advice for a test instance. It does this by utilizing both density values and selected layer combinations as seen in (5) and (6) respectively (see Section 4.4 for more information). If an alarm is triggered, (7) this test instance is fed into the trained DCGAN to generate a transformed instance, which is (8) then input into M to obtain the alternative prediction.

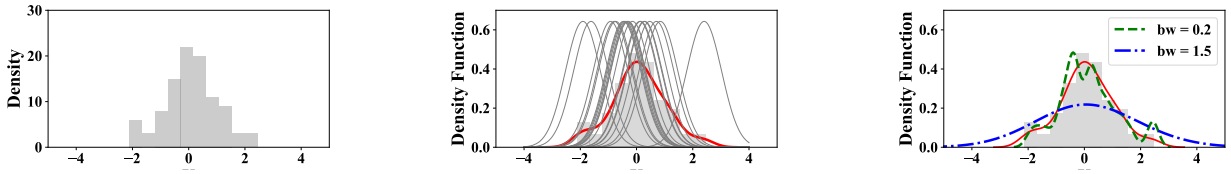
Next we detail each step in SelfChecker++.

4.1 KDE of the Training Set

Let M, L and C represent a trained classifier, number of layers (excluding the input layer) and number of classes respectively. Moreover, with respect to training dataset D_{train}, let the inputs $\mathcal{X}^t$ and ground truth labels $\mathcal{Y}^t$ be represented as $\{\mathbf{x_1}, \ldots, \mathbf{x_n}\}$ and $\{y_1, \ldots, y_n\}$ respectively. The same is also set for validation dataset D_{valid} where $\mathcal{X}^v$, $\mathcal{Y}^v$, and $\hat{\mathcal{Y}}^v$ represent inputs, ground truth labels and classifier predictions respectively.

Furthermore, following the execution of M on an instance, the feature vectors are attainable. The feature vectors

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3200421



(a) Histogram (b) KDE (c) Bandwidth

Fig. 4. An example to illustrate KDE computation with *(a)* showing the set of input 1D points, *(b)* showing how to obtain the distribution using a KDE, and *(c)* showing the distributions obtained by using different bandwidths.

of every layer (i.e., layer output from training dataset) are represented as $\mathcal{V}^t = \{\mathbf{v}_1^t, \ldots, \mathbf{v}_L^t\}$ where $\mathbf{v}_l^t \in \mathbf{R}^{n_l}$ and n_l represents the number of neurons at layer l. Usually, it is observed that the attention of features varies depending on the class and layers. The goal of SelfChecker++ is, whilst utilizing D_{train} and its feature vectors, to calculate the per layer and per class density probabilities. Once the density probabilities are attained, SelfChecker++ uses them to approximate the similarity between the feature vectors at a particular layer based on an input and the feature vectors at the same layer based on the training dataset.

Utilizing a finite amount of samples from a given population, one is able to approximate the probability density function (PDF) using KDE [35]. This is because KDE does not make any distribution assumptions and is non-parametric. With the PDF, approximation on a random variable's relative likelihood is straight forward. Here, we utilize a Gaussian kernel because it has good performance when it comes to multivariate data (which most datasets inadvertently are). In addition, it also outputs smooth functions. With samples $\{x_1, x_2, \ldots, x_m\}$, SelfChecker++ approximates the KDE function f with the following:

$$\hat{f}(x) = \frac{1}{mh} \sum_{i=1}^{m} K(\frac{x - x_i}{h}) \tag{3}$$

where K and h represents the Gaussian kernel function and *bandwidth* respectively.

To get an idea of how this function operates, consider Fig. 4. Initially, every single observation is fitted with a Gaussian curve at it's center. Naturally, this is the kernel. These curves are then added to calculate the density value per point. In Fig. 4(b), the red curve depicts the normalized curve with an area under the curve of 1. Moreover, h determines the tightness of the approximation onto the samples. It can be thought of as the kernel's width. From Fig. 4(c), it can be observed that the bigger h is, the curve becomes smoother and flatter. In contrast, the smaller h becomes, the curve becomes rougher and sharper. Lastly, the decision on $h's$ value is determined by the amount of samples as well as their dimensions.

For every class at every layer, SelfChecker++ utilizes the Gaussian KDE to approximate the PDF that the training data for a particular class induces on the feature vector of a layer. After the PDFs are attained, with a test instance, SelfChecker++ is able to approximate the probability densities for every class at every layer. Lastly, SelfChecker++ utilizes these probability densities for layer inference:

Definition 1 (Inferred class for a layer). Given a test instance, the *inferred class for layer* l is the class for which the test instance induces the maximum estimated probability density among l's per-class density functions.

Algorithm 1: KDE Estimation and Inference

Input: Input instances in D_{train}, D_{valid}: $\mathcal{X}^t$, $\mathcal{X}^v$, true labels in D_{train}: $\mathcal{Y}^t$;
Trained model M with L layers and C classes;
Variance threshold: t_{var}
Output: KDE functions for each combination of class and layer: $kdes$;
Inferred classes for all layers on D_{valid}: $kdeInferL^v$

1 # Estimation
2 **for** *c in C* **do**
3 Obtain instances $\mathcal{X}_c^t$ whose true label is c;
4 **for** *l in L* **do**
5 $\mathbf{v}_{lc}^t = M.output_l(\mathcal{X}_c^t)$;
6 Remove elements in $\mathbf{v}_{lc}^t$ whose variance is less than t_{var};
7 $\hat{f}(x) = \frac{1}{|\mathbf{v}_{lc}^t|h} \sum_{i=1}^{|\mathbf{v}_{lc}^t|} K(\frac{x - \mathbf{v}_{lc}^t[i]}{h})$;
8 $kdes[l][c] = \hat{f}(x)$;
9 **end**
10 **end**
11 # Inference
12 **for** $\mathbf{x}$ *in* $\mathcal{X}^v$ **do**
13 **for** *l in L* **do**
14 $\mathbf{v}_l = M.output_l(\mathbf{x})$;
15 Remove values of the neurons filtered in the training set from $\mathbf{v}_l$;
16 **for** *c in C* **do**
17 $kde\_values[c] = kdes[l][c](\mathbf{v}_l)$;
18 **end**
19 $kdeInferL^v[\mathbf{x}.index][l] = max(kde\_values).index$;
20 **end**
21 **end**

The steps for KDE estimation as well as inference is shown in Algorithm 1. Firstly, for every class at every layer, their density distribution functions with respect to feature vectors of D_{train} are found. Specifically, this happens in Lines 1-10 where a Gaussian KDE was utilized when extrapolating them. Previously demonstrated in Fig. 1, our aim is to extrapolate the attention patterns with respect to the input. Moreover, an important factor to consider is that the performance of an instance at a layer varies from class to class. For example, in Fig. 1, it is observed that the heatmaps at the fist convolutional layer are different for both digits 3 and 6. In addition, the heatmaps for digit 6 differ between the fist and second convolutional layers as well. Because of this, in Line 3, the training dataset D_{train} undergoes class-level partitioning. Following which, SelfChecker++ then attains the outputs at every layer for a class in Line 5. In order to reduce the dimensions, two techniques are employed. In particular, mean-pooling and variance thresholding as shown in Line 6. Mean-pooling is utilized on the convolutional layers and neurons with values of variance less than t_{var} are discarded. This is because these neurons have minimal impact to the KDE. Lastly, in Line 7,

the feature vectors which conform to t_{var} are utilized when extrapolating the per class and per layer density functions. In Line 8, these density functions are stored for future use (e.g., Line 17).

Given an inference instance, the per layer outputs are attained in Line 14. Following that, in Line 15, the neurons discarded previously in Line 6 are similarly discarded. In Line 17, SelfChecker++ approximates the per class density values using density functions $kdes$ previously attained in Line 8. Line 19 shows how the inference for a layer is calculated. It is basically the class with the highest density value. This shows that the feature vectors of the instance at the layer in question are similar to the feature vectors from D_{train} that belong to the class with the highest density value. An example of this is shown in Fig. 1 where the inference for digit 3 is 3 for the first layer, 3 for the second layer and 2 for the third layer.

4.2 Layer Selection

In Section 3 it was observed that attentions vary from layer to layer. Moreover, some of these attentions might be incorrect. One such scenario is Fig. 1 where for digit 6, there are two layers that disagree with the final prediction. If SelfChecker++ chooses to evaluate the outputs of the layers in this example, it is most likely that SelfChecker++ would agree with the fact that the model's prediction is of low confidence. As such, it is key to establish a *robust layer selection* technique in order to trigger alarms accurately.

Firstly, we shall define precisely what the *confidence* of a prediction represents. The definition stemmed from this particular observation: given a test instance, if the inferred classes of DNN layers are different from the final prediction, then the decision made by the model on the test instance will tend to be incorrect. This can be observed in Fig. 1, where for digit 6, the last two layers disagree with the prediction of 1. We utilized the following techniques when evaluating our observation: Spearman rank-order correlation coefficient and p-values [36]. More specifically, Spearman rank-order measures the relationship between the prediction correctness and the consistency of inferred layer classes and final predictions. Generally, a p-value of 0.05 (5%) or less is considered statistically significant [36]. As shown in Table 1, the values indicate that, for all dataset and model combinations, p-value $<< 0.05$.

TABLE 1
p-value

p-value	ConvNet	VGG-16	ResNet-20
MNIST	7.38e-62	4.61e-105	3.09e-26
FMNIST	0.0	9.93e-251	1.42e-281
CIFAR-10	0.0	1.71e-267	1.03e-232
CIFAR-100	0.0	0.0	0.0

In equation (4), the confidence δ with respect to a model's prediction is formally defined.

$$\delta = \frac{N_{kdeInferL_x == \hat{y}}}{N_{selectedLayerC_{alarm}[\hat{y}]}} \tag{4}$$

Here, $\mathbf{x}$ and $\hat{y}$ refer to the test instance and model's prediction respectively. Moreover, $N_{kdeInferL_x==\hat{y}}$ and $N_{selectedLayerC_{alarm}[\hat{y}]}$ refer to the number of selected layers. The former are the ones which agree with $\hat{y}$ whilst the latter are the ones specifically for class $\hat{y}$. From the concept of maximum voting, when $\delta < 0.5$, it is said that the model's prediction is of *low confidence*.

Algorithm 2: Layer Selection for Alarm

Input: Input instances in D_{valid}: $\mathcal{X}^v$, true labels and predictions: $\mathcal{Y}^v$, $\hat{\mathcal{Y}}^v$;
Total classes: C;
Inferred classes for all layers on D_{valid}: $kdeInferL^v$
Output: Selected layers for all classes: $selectedLayerC_{alarm}$

1 **for** *c in* C **do**
2 Obtain the indexes idx_c of instances $\mathcal{X}^v_c$ whose prediction $\hat{\mathcal{Y}}^v$ is c;
3 Generate all kinds of layer combinations $combL$;
4 **for** *layers* l_s *in* $combL$ **do**
5 **for** *l in* l_s **do**
6 y_s.add($kdeInferL^v[idx_c][l]$);
7 **end**
8 $KdePredPos$.add(index of $sum(y_s != \hat{\mathcal{Y}}^v[idx_c]) >= sum(y_s == \hat{\mathcal{Y}}^v[idx_c])$);
9 $TrueMisBehavior$.add(index of $\hat{\mathcal{Y}}^v[idx_c] != c$);
10 $TP = TrueMisBehavior$ & $KdePredPos$;
11 $FP = \neg TrueMisBehavior$ & $KdePredPos$;
12 $FN = TrueMisBehavior$ & $\neg KdePredPos$;
13 $F1 = 2 * TP/(2 * TP + FN + FP)$;
14 **if** *F1 is max* **then**
15 $selectedLayerC_{alarm}[c] = l_s$;
16 **end**
17 **end**
18 **end**

Algorithm 2 details the layer selection process for every class. The goal when it comes to layer selection here is to attain high alarm accuracies. As aforementioned in Algorithm 1, the training dataset was utilized in order to approximate the density functions which allowed us to infer classes for every layer with respect to an instance. Previously discussed in Section 3, attentions vary from layer to layer and might be deceptive at times. Hence, validation dataset D_{valid} is utilized in order to choose layers. With D_{valid}, SelfChecker++ partitions the data with respect to their predictions as shown in Line 2. Following which in Lines 4-17, for every class, every layer combination is tested in order to locate the combination that attains the best accuracy. In order to achieve this, for a specific combination, the inferred class for every layer within the combination is attained with the aid of $kdeInferL^v$ at Line 6. In particular, $kdeInferL^v$ refers to the KDE inferences attained in Algorithm 1. In order to check if M provides an incorrect prediction for an instance, SelfChecker++ looks at every layer in the specific combination. In the case where there are more layers disagreeing with the model's prediction than there are agreeing (i.e., $\delta < 0.5$), the model's prediction is then said to be incorrect as detailed in Line 8. A layer disagrees with the model's prediction when it's inferred class is not that of the prediction. Furthermore, if this happens and the prediction is not the same as the true label, the alarm is correct and this indicates a True Positive. In contrast, if the prediction is the same as the true label, the alarm is wrong and this indicates a False Positive instead. The F1-score is utilized in order to calculate the accuracy of alarm, detailed in Line 13. The layer combination l_s which attains the best F1-score would be the selected layers with respect to the current class, detailed in Line 15.

Similarly, there is also a need to find the optimal layer combinations when providing advice once an alarm is trig-

Algorithm 3: Layer Selection for Advice

Input: Input instances in D_{valid}: $\mathcal{X}^v$, true labels and predictions: $\mathcal{Y}^v$, $\hat{\mathcal{Y}}^v$;
Total classes: C;
Inferred classes for all layers on D_{valid}: $kdeInferL^v$;
Selected layers for all classes: $selectedLayerC_{alarm}$
Output: Selected layers and weights per class: $selectedLayerPosC_{advice}$, $\mathbf{W}_{pos}$, $selectedLayerNegC_{advice}$, $\mathbf{W}_{neg}$;

1 **for** c_p *in* C **do**
2 Obtain the indexes idx_{c_p} of instances $\mathcal{X}^v_{c_p}$ whose prediction $\hat{\mathcal{Y}}^v$ is c_p;
3 Generate y_s given $selectedLayerC_{alarm}[c_p]$;
4 Generate all kinds of layer combinations $combL$;
5 $KdePredPos$.add(index of $sum(y_s$!= $\hat{\mathcal{Y}}^v[idx_{c_p}]) >= sum(y_s == \hat{\mathcal{Y}}^v[idx_{c_p}]))$;
6 $TrueMisBehavior$.add(index of $\mathcal{Y}^v[idx_{c_p}]! = c_p)$;
7 $FP = \neg TrueMisBehavior$ & $KdePredPos$;
8 **for** c_t *in* C **do**
9 idx_{c_t}.add(index of $KdePredPos$ where $\mathcal{Y}^v[KdePredPos] = c_t)$;
10 Select layers $selectedLayerPosC_{advice}$ with highest accuracy acc_{max} from $combL$;
11 **if** $c_t = c_p$ **then**
12 $\mathbf{W}_{pos}[c_p][c_t] = len(idx_{c_t}) * acc_{max}/len(KdePredPos)$
13 **else**
14 $\mathbf{W}_{pos}[c_p][c_t] = len(idx_{c_t}) * acc_{max}/(len(KdePredPos) - FP)$
15 **end**
16 **end**
17 $KdePredNeg$.add(index of $sum(y_s$!= $\hat{\mathcal{Y}}^v[idx_{c_p}]) < sum(y_s == \hat{\mathcal{Y}}^v[idx_{c_p}]))$;
18 $TN = \neg TrueMisBehavior$ & $KdePredNeg$;
19 Iterate Lines 8-16 to obtain $selectedLayerNegC_{advice}$ and $\mathbf{W}_{neg}$
20 **end**

gered. Algorithm 3 shows the steps taken by SelfChecker++ in order to find these optimal combinations so as to attain the highest advice accuracy. In Line 2, D_{valid} is partitioned C times. Following which, for every partition, the best combination is found. Utilizing $selectedLayerC_{alarm}$ from Algorithm 2, in Line 3 based on the current class c_p, the selected layer's inferred classes are calculated similar to Line 6 of Algorithm 2. And like Algorithm 2, in Line 5 of Algorithm 3, if $\delta < 0.5$, the model is said to have misbehaved. In Lines 9-10, the best layer combination is found. More specifically, it is where the instance with label c_t is predicted as c_p. From the fact that not every class has correlations, for each class combination, weights are attained in Lines 12 and 14. A scenario illustrating this is that digit 1 is misclassified as digit 7 much more than it is misclassified as digit 2. Finally in Lines 17-19, the combination of layers which attains the best accuracy is found. More specifically, this is on the situation where the layers selected from Algorithm 2 exhibit a negative decision (i.e., whereby the model behaved normally).

Boosting strategy: Utilizing the selected layers in Algorithm 2, both positive and negative decisions which are made by these layers are considered by SelfChecker++. This is imperative as it raises the alarm's quality. More specifically, if these selected layers suggest triggering an alarm but advice from $selectedLayerPosC_{advice}$ in Line 10 has the same class with the prediction, an alarm is ultimately not triggered. Conversely, if these layers suggest that the prediction is correct but advice from $selectedLayerNegC_{advice}$ in Line 19 is not the same as the prediction, an alarm is ultimately triggered.

4.3 GAN-based Transformation

To "denoise" the unintentional abnormal and intentional adversarial examples, we train a DCGAN on the training dataset to generate images similar to these examples but conform to the distribution of the training data. To train the DCGAN, random vectors are fed into the generative net G to generate images with similar distributions, while the discriminator D classifies the generated images as real or fake. G and D are trained in an adversarial fashion to minimize the loss function (Equation 2 in Section 3.2). The aim is to gradually improve images generated by G until they successfully fool D (i.e., D trains to differentiate generated and real images).

Since we want to design a general transformation framework for all DNN models, it would be ideal if the DCGAN-based transformation approach shared some connections (e.g., architecture) with the original DNN model to remove the overhead of searching for the optimal generator and discriminator combinations. This is because the search is notoriously difficult for GANs. Furthermore, there is a need to maintain fairness by ensuring that any improvement in accuracies from SelfChecker++ are not due to architectural differences. We thus design the generative model G and the discriminative model D to follow a similar architectures as the original DNN model.

When an alarm is triggered by a given test instance, SelfChecker++ uses a trained DCGAN to synthesize a semantically-preserving input w.r.t the given instance from the latent space. It uses this generated input instead of the original input for the DNN model M during deployment.

We propose to use the representative power of the DCGAN trained in the training phase so as to diminish the effect of noise perturbations that lead to wrong predictions. The DCGAN projects the input instance x onto a range of the DCGAN's generator G to generate $x\prime$ before feeding it to the deployed DNN M.

To preserve the semantics of x, SelfChecker++ needs to find a proper latent vector z such that $G(z)$ is close to x. We use gradient descent to find this z with a minimization:

$$min_z||G(z) - x||^2 \tag{5}$$

Similar to prior work [37], we approximate the above formulation with L gradient descent steps and use R multiple random starts to search in $Z_0 = \{\mathbf{z_0}^1, \ldots, \mathbf{z_0^R}\}$ so as to minimize (5). Based on the selected z^*, the transformed instance will be $x\prime = G(z^*)$.

Projecting the original instance onto the range of G can have the desirable effect of reducing the noise perturbation and making the sample conform further to the distribution of training data. This helps to make the model more robust to crafted adversarial samples. The generated $x\prime$ is then fed into the deployed DNN M instead of the original instance x, and the output of M is returned as the alternative prediction.

4.4 Checking the Model in Deployment

Algorithm 4 details the steps SelfChecker++ takes when checking a deployed network. Based on a test instance, should SelfChecker++ deem the network's prediction as incorrect, an alarm would be triggered and advice would then be given.

Algorithm 4: Checking Model in Deployment

Input: Input instance and its prediction by M with L layers: $\mathbf{x}$, $\hat{y}$;
KDE functions for all layers and classes: $kdes$;
Selected layers for all classes: $selectedLayerC_{alarm}$, $selectedLayerPosC_{advice}$, $selectedLayerNegC_{advice}$;
Weights for advice: $\mathbf{W}_{pos}$, $\mathbf{W}_{neg}$;
Trained generator in WGAN: G;
Gradient descent steps and number of random starts: L, R
Output: $alarm$ and $advice$ $c\prime$

1 Generate inferred class for each layer $kdeInferL$ using KDE functions $kdes$;
2 $L_{alarm} = selectedLayerC_{alarm}[\hat{y}]$;
3 Generate y_s given L_{alarm} and $kdeInferL$;
4 **if** $sum(y_s != \hat{y}) >= sum(y_s == \hat{y}))$ **then**
5 initialize $prob$ with C dimensions;
6 **for** c *in* C **do**
7 $L_{advice} = selectedLayerPosC_{advice}[\hat{y}][c]$;
8 **for** l *in* L_{advice} **do**
9 $prob[c] = sum(kdeInferL[l] == c)$;
10 **end**
11 $prob[c] = prob[c] * \mathbf{W}_{pos}[\hat{y}][c]/len(L_{advice})$
12 **end**
13 $advice = max(prob[c]).index$;
14 **if** $advice$ *!=* $\hat{y}$ **then**
15 $alarm = True$;
16 $Z_0 = random\_initializer(\{\mathbf{z}^1, \ldots, \mathbf{z}^{\mathbf{R}}\})$;
17 **for** z^i *in* Z_0 **do**
18 **while** $e < L$ **do**
19 $e = e + 1$;
20 $z_e^i = z_{e-1}^i - \gamma \nabla ||G(z_{e-1}^i) - \mathbf{x}||^2$;
21 **end**
22 $Z_L = \{\mathbf{z}^1, \ldots, \mathbf{z}^{\mathbf{R}}\}$;
23 **end**
24 $z^* = argmin_{z \in Z_L} ||G(z) - \mathbf{x}||^2$;
25 $c\prime = M.output(G(z^*))$;
26 **else**
27 $alarm = False$
28 **end**
29 **else**
30 Iterate 5-28 if the alarm is not triggered initially;
31 **end**

Algorithm 1 provides $kdes$ (i.e., the KDE functions for every layer and class combination) which Algorithm 4 then uses in conjunction with the current layer outputs to attain, for every layer, the inferred classes for the current test instance, $kdeInferL$. Following which, in Lines 2-3, SelfChecker++ attains y_s which is basically the inferred classes for layers in L_{alarm} (i.e., layers selected with respect to the prediction). In Line 4, if it is seen that amongst the inferred classes in y_s, class $\hat{y}$ is not of the majority, SelfChecker++ raises an initial alarm which is still subjected to the **Boosting strategy** (i.e., where the alarm might not be raised eventually).

Furthermore as detailed in Lines 5-12, with the aid of $selectedLayerPosC_{advice}[\hat{y}]$ and $\mathbf{W}_{pos}$, the per class weighted probabilities are attained. In Lines 13-25, an alarm is raised should $\hat{y}$ continue to differ from the class with the highest probability. $x\prime$, a semantically preserving input which conforms more towards the training data distribution is then found with (5) from Section 4.3. In contrast, detailed in Line 27, an alarm is not raised if $\hat{y}$ does not differ from the class with the highest probability. Lastly, in the case where an alarm is not initially raised in Line 4, a strategy similar to the aforementioned one will be utilized as well.

5 Evaluation

In this section, we show the effectiveness and efficiency of SelfChecker++ with experiments. We first list out the research questions we aim to address.

5.1 Research Questions

RQ1. Alarm Accuracy: *How effective is SelfChecker++ in detecting DNN misclassifications in deployment?*

In order to access SelfChecker++ with respect to raising alarms on unintended anomalies during deployment, we use the test dataset and calculate the alarm accuracy to compare with similar applications including, SelfOracle [14], ConfidNet [13], and Dissector [15]. With regards to the comparison, the VAE (variational autoencoder) which is a variant from SelfOracle was chosen as it achieved the best performance amongst other variants, with a confidence threshold of 0.05. Moreover as ConfidNet's threshold for failure prediction was unknown, we found the best threshold between $0 - 1$ which had the highest F1-score for ConfidNet from the validation dataset. Similarly, to distinguish beyond-inputs from within-inputs by Dissector, we found the best threshold which had the highest F1-score from the validation dataset. Furthermore for Dissector, the validation dataset was also used to find the best weight growth type (i.e., *linear*, *logarithmic* or *exponential* [15]) which had the highest Area Under Curve (AUC) for each DNN classifier and dataset.

RQ2. Detection of Adversarial Examples: *How is the effectiveness of SelfChecker++ in detecting adversarial examples in deployment?*

In order to address this research question, we calculate the alarm accuracy of SelfChecker++ on test and adversarial examples generated by RobOT [38], ADAPT [39], FGSM [26], and PGD [40] respectively. In order to judge the effectiveness of SelfChecker++, we compare its alarm accuracy with Dissector [15] that claims its usefulness in detecting adversarial examples. Note that the instances in the test dataset wrongly predicted by DNN models are regarded as adversarial samples to answer this research question. For fair comparisons, we deal with Dissector the same as in RQ1. We also check ConvNet on MNIST and FMNIST, and Resnet-20 on CIFAR-10, which are the conditions used in [38], [39].

RQ3. Advice Accuracy: *How does SelfChecker++ compare to SelfChecker on the advice accuracy?*

Apart from raising alarms, there is also a need to then determine if the advices provided by SelfChecker++/SelfChecker are accurate. They both use same alarm mechanism but different advice mechanisms. We thus compare their final advice accuracies on test and adversarial examples to check the quality of advice provided by SelfChecker++ against that by SelfChecker and the accuracy of M (i.e., the original DNN model). Other techniques mentioned previously are not designed to provide advice.

RQ4. GAN-based Transformation: *Can GAN-based transformation be used on the entire dataset instead of combined with the alarm analysis in SelfChecker++?*

As discussed in Sections 4.3, we use a GAN-based transformation to transform the inputs triggering alarms into semantically-preserving inputs that conform to the distribution of the training data. These transformed inputs will then be passed through the original DNN model M, attaining alternative predictions. However, whether this transformation

TABLE 2
DNN models and datasets used in the experiments.

Dataset	# Class	# Train	# Valid	# Test	DNN models					
					ConvNet		VGG-16		ResNet-20	
					# Layers	Accuracy%	# Layers	Accuracy%	# Layers	Accuracy%
MNIST	10	50,000	10,000	10,000	8	99.36	16	98.87	20	99.46
FMNIST	10	50,000	10,000	10,000	8	92.13	16	93.75	20	92.74
CIFAR-10	10	40,000	10,000	10,000	8	80.45	16	92.17	20	92.08
CIFAR-100	100	40,000	10,000	10,000	8	44.04	16	66.79	20	69.52

DAVE-2 and Chauffeur for self-driving cars are regression models so we exclude them in this table.

can be used to all inputs without the alarm mechanism to improve accuracy is unknown. In order to address this, M's accuracy is compared on two types of datasets where all test inputs are transformed by the GAN versus only those that trigger an alarm.

RQ5. Deployment Time: *What is the time overhead of SelfChecker++ in deployment for a given test instance?*

Taking note of the specifics of different algorithms during deployment, we access their respective component's computation time during deployment. With the aid of two DNNs, ConfidNet attains the output. SELFORACLE on the other hand utilizes a reconstructor for the loss and anomaly detector. DISSECTOR generates probability vectors and performs validity analysis[2]. SelfChecker and SelfChecker++ perform DNN-based computations, KDE-based inferences, alarm analysis and advice generation. But they use different advice mechanisms. We will also compare their deployment time.

RQ6. Layer Selection: *Does the choice of layers for selection by SelfChecker++ have an impact on its alarm accuracy?*

Cases where DNNs are able to attain the correct prediction in layers prior to the final layer, i.e., "over-thinking", is described as a prevalent weakness of DNNs by Kaya et al. [41]. As aforementioned in Section 3, this can have a negative effect as the correct prediction can then change to the wrong prediction in the final layer. As such, it is of great importance to select the most appropriate layers for each class. In order to access the influence layer selections had on the accuracy of the alarm, we implemented three selection strategies (see Section 5.3: RQ6).

RQ7. Boosting Strategy: *Does the boosting strategy improve SelfChecker++'s alarm accuracy, particularly in terms of decreasing the number of false alarms?*

As aforementioned in Sections 4.2 and 4.4, we employed a boosting strategy in order to determine if an alarm should be raised.

5.2 Experimental Setup

We evaluated SelfChecker++ on the following datasets - MNIST [42], FMNIST [43], CIFAR-10 [44] and CIFAR-100 [44]. During the evaluation, we made use of three commonly-used DNN architectures as well - ConvNet [45], VGG-16 [46] and ResNet-20 [47]. Moreover, in the constraints of self-driving car scenarios that were tested on NVIDIA's DAVE-2 [7] and Chauffeur [48], we compared SelfChecker++ against SELFORACLE [14] in terms of alarm accuracy. To reduce the possibility of fluctuation caused by randomness, we performed our experiments three times and present an average of the three results. One exception to this, however, is that we performed the experiments on the driving datasets just once as we were using the pre-trained models which SELFORACLE's [14] authors released.

5.2.1 Datasets and DNN models

Table 2 depicts our datasets and their respective parameters (number of classes and dataset size) and their performance (testing accuracy) on various DNN architectures. These are commonly-used image datasets and DNN architectures. The DNNs include both small and large architectures with layers ranging from 8 to 20. The accuracies shown in Table 2 are similar to that of the state-of-the-art. There are two modules in SelfChecker++ (Section 4). The training module utilizes the training and validation datasets whereas the deployment module utilizes the test dataset when assessing the performance of SelfChecker++.

In experiments involving the driving datasets, the datasets and models were shared by the authors of SELFORACLE. Both DNN models had 37,947 and 9,486 training and validation images, respectively. For testing, DAVE-2 [7] had 134,820 images while Chauffeur [48] had 250,830 images. The reason as to why the testing images differ is because self-driving cars (with the DNN models) gathered them. Moreover as collisions and out-of-bound scenarios halts the process, the number of images collected in the end are bound to be different for both models. For their architectures, DAVE-2 and Chauffeur have five and six convolutional layers respectively. With regards to fully-connected layers, DAVE-2 and Chauffeur have three and one respectively.

5.2.2 Configurations

As mentioned in Section 4, all of the neurons which had activation values with a variance less than t_{var} in Algorithm 1 were filtered out. This is because they do not contribute significantly to the KDE. The setting for all of the RQ had a default variance threshold of 10^{-5}. Furthermore, utilizing Scott's Rule [49], the amount of data points and dimensions, the KDE's bandwidth was set accordingly. Gradient descent steps (L) and the number of random starts (R) in Algorithm 4 are used to approximate the formulation (5) in Section 4.3 to search for z^*. The default value of L was set to

2. It should be noted that Wang et al. [15] only incorporated validity analysis. However, being the input to validity analysis, we think that probability vector generation should also be done during deployment.

TABLE 3
Alarm accuracy on unintended anomalies.

Dataset	DNN	↑ TPR %				↓ FPR %				↑ F1 %			
		SO	*DT*	*CN*	*SC++*	*SO*	*DT*	*CN*	*SC++*	*SO*	*DT*	*CN*	*SC++*
MNIST	*ConvNet*	18.75	60.94	60.94	**62.50**	4.39	0.24	0.58	**0.23**	4.69	61.42	48.45	**62.99**
	VGG-16	20.35	68.14	61.95	**74.34**	4.29	0.32	0.46	**0.31**	8.21	69.37	61.40	**73.68**
	ResNet-20	33.33	46.30	50.00	**59.26**	4.45	0.39	0.50	**0.32**	6.99	42.37	41.22	**54.24**
FMNIST	*ConvNet*	9.53	**47.65**	38.12	41.55	5.60	4.03	0.73	**0.51**	10.89	48.92	51.99	**56.33**
	VGG-16	8.00	**48.48**	43.36	46.88	5.75	4.16	0.98	**0.86**	8.24	45.98	54.86	**58.66**
	ResNet-20	9.64	**54.96**	47.66	51.79	5.69	3.76	1.14	**0.98**	10.57	54.14	58.74	**63.03**
CIFAR-10	*ConvNet*	5.01	61.43	58.57	**61.89**	3.97	9.83	2.29	**2.04**	8.26	60.86	69.73	**72.69**
	VGG-16	6.39	**53.77**	43.17	49.30	3.94	4.47	3.03	**1.16**	8.36	52.10	48.29	**60.50**
	ResNet-20	7.07	47.98	49.87	**52.15**	3.96	4.93	1.03	**0.64**	9.23	46.74	61.62	**65.35**
CIFAR-100	*ConvNet*	30.20	73.20	82.49	**92.32**	26.98	**18.57**	63.90	48.46	39.89	77.94	70.87	**80.12**
	VGG-16	10.48	82.78	78.20	**84.22**	7.88	23.78	16.17	**6.57**	16.59	71.79	74.22	**85.31**
	ResNet-20	11.25	75.16	61.15	**80.97**	7.64	21.63	13.56	**7.09**	17.49	66.96	63.67	**82.14**
Driving	*DAVE-2*	76.85	-	-	**99.01**	**7.29**	-	-	9.37	46.43	-	-	**49.88**
	Chauffeur	81.15	-	-	**93.44**	4.77	-	-	**4.56**	32.25	-	-	**37.25**

SO, DT, CN, and SC++ stand for SELFORACLE, DISSECTOR, ConfidNet, and SelfChecker++, respectively.

200 and R was set to 2. We selected these default values by balancing the time consumption and accuracy. Increasing them led to an exponential increase in time consumption with only a minor improvement in accuracy. All of our experiments were done on an Ubuntu 18.04 server with an Intel i9-10900X (10-core) CPU @ 3.70GHz, RTX 2070 SUPER GPU, and 64GB RAM.

5.2.3 Metrics

SelfChecker++ provides an alarm once it finds that most of the selected layers do not agree (i.e., their KDE inferences do not agree) with the output of the model. For measurement, we utilize the standard confusion metrics: True Positive (TP), False Positive (FP), True Negative (TN) and False Negative (FN). A TP instance is where SelfChecker++ sets off an alarm when the model's output is indeed wrong. Conversely, a FN instance is where there is no alarm from SelfChecker++ when the model's output is indeed wrong. FP is for when SelfChecker++ wrongly raises an alarm and TN is for when there isn't an alarm raised by SelfChecker++ on correct classifications. Our goals are threefold - a high true positive rate (TPR = TP / (TP+FN)), a low false positive rate (FPR = FP / (TN+FP)) and a high F1-score (F1 = (2 * TP) / ((2 * TP) + FN + FP)).

5.3 Results and Analyses

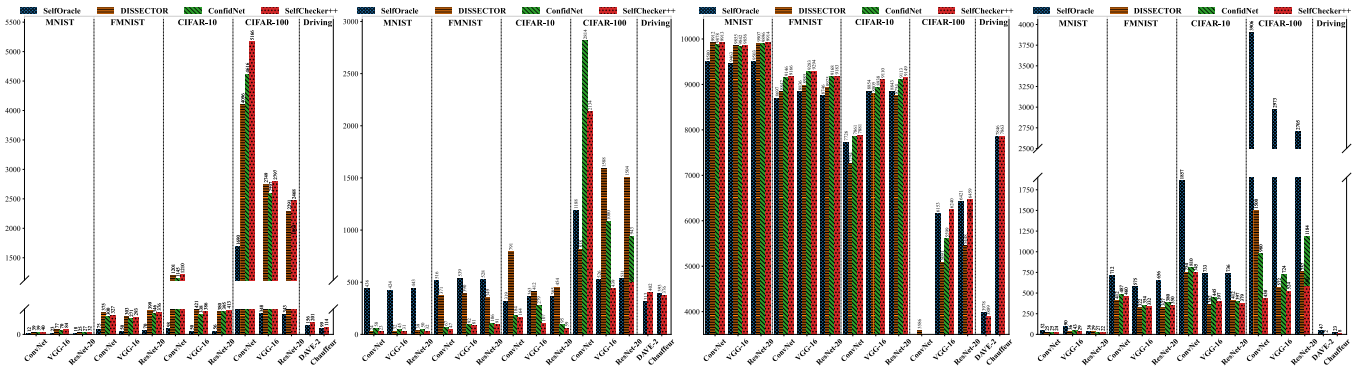
This subsection now discusses our experimental results and the research questions in detail.

5.3.1 RQ1. Alarm Accuracy

The TPR, FPR and F1-score for SELFORACLE, ConfidNet, DISSECTOR and SelfChecker++ on each model and dataset during deployment are depicted in Table 3. In addition, the TP, FP, TN and FN metrics with respect to each model and dataset are shown in Fig. 5. It is observed that compared to SELFORACLE and ConfidNet, SelfChecker++ comes out ahead in triggering correct alarms (TP) and misses fewer true alarms (FN).

Considering average TPR on traditional DNN classifiers, SelfChecker++, SELFORACLE, ConfidNet and DISSECTOR attains 63.10%, 14.17%, 56.30% and 60.06% respectively. This means that for over half of the misclassifications, SelfChecker++ correctly raises an alarm and outperforms the rest. More specifically, the highest TPR attained by SelfChecker++ was 92.32% (i.e., SelfChecker++ found over 90% of the misclassifications). It should be noted that DISSECTOR does attain a better TPR for four scenarios. This can be attributed to the fact that like SelfChecker++, DISSECTOR takes advantage of the internal layer features. In particular, it uses multiple sub-models retrained on top of the internal layers which could cause it to learn some information which SelfChecker++ does not. Nonetheless, as SelfChecker++ generally is better than DISSECTOR in TPR, the extra information DISSECTOR has is limited to a certain extent. Furthermore, SelfChecker++ outperforms both SELFORACLE and ConfidNet in terms of TPR over every dataset and DNN combination. Since SELFORACLE does not have internal information and ConfidNet looks at high-level representations, we conclude that for the purpose of identifying misclassifications, internal layer features are key. Considering FPR, SelfChecker++ attains the lowest amongst all the competitors except DISSECTOR checking ConvNet on CIFAR-100. Having a lower FPR basically means that SelfChecker++ sets off fewer false alarms. This is not surprising as the boosting strategy mentioned in Section 4.2 allows SelfChecker++ to be cautious when setting off alarms.

Lastly, SelfChecker++ achieves the highest F1-score for all dataset and DNN combinations averaging at 67.92% against 12.45%, 58.77%, and 58.22% for SELFORACLE, ConfidNet, and DISSECTOR, respectively. The reason SELFORACLE exhibits poor accuracy on the traditional DNN classifiers stems from the fact that it is primarily made for time series analysis. More specifically, for video frame sequences which inadvertently do not differ much over short time spans. ConfidNet on the other hand suffers from overfitting which would bring about performance limitations. The original DNN model (which ConfidNet utilizes as a base for it to be trained on top of) has its weights frozen and has a loss based on a class's true probability. Thus, not having a significant amount of incorrect predictions from the training dataset after training the original DNN model leads to over-



(a) # True Positives (TP) (b) # False Positives (FP) (c) # True Negatives (TN) (d) # False Negatives (FN)

Fig. 5. Confusion metrics comparing the performance of all approaches.

fitting. Moreover, as mentioned earlier in Section 5.2, due to the fact that we consider wrong predictions as positive cases and [13] instead considers correct predictions as positive cases, the ConfidNet percentages in Table 3 do not match the ones presented in [13].

With regards to the self-driving car experiments, we modified the prediction of steering angles via regression into binary classification of steering angles (i.e., normal steering angle or anomalous steering angle). ConfidNet was not used in the self-driving car experiments as true class probability was its base. Similarly, DISSECTOR was not used as both the first and second highest class probabilities were required. With the validation dataset, we fit a distribution (specifically Gamma) to the MSE of predictions and real-valued angles, and the per layer density values (attained from Algorithm 1). Taking ϵ from the distribution as 0.05 (similar to SELFORACLE), if the error for a single instance is more than 0.05, it would then be categorized as an anomaly. The same also goes for the density values which would be predicted as an anomaly if less than 0.05. SelfChecker++ is then utilized to solve the problem via binary classification. Comparing SelfChecker++ and SELFORACLE in Table 3, SelfChecker++ attains a better TPR across both driving models which means that SelfChecker++ sets off more correct alarms. Furthermore, although SelfChecker++ is seen to set off more false alarms with respect to DAVE-2, it sets off more true alarms than SELFORACLE (201 as opposed to 156), missing only 2 of them. SelfChecker++ also outperforms SELFORACLE on DAVE-2 and Chauffeur in terms of F1-score.

For RQ1, we showed that SelfChecker++ effectively triggers alarms that predict misbehaviors of DNN models in deployment with high TPR, low FPR and high F1-score.

5.3.2 RQ2. Detection of Adversarial Examples

Table 4 compares the accuracies of detecting adversarial examples by DISSECTOR and SelfChecker++. We did not use SELFORACLE and ConfidNet as baselines since these tools were not designed to handle adversarial examples. By contrast, DISSECTOR discussed its efficiency on identifying adversarial examples with higher and more stable AUC values than all variants of mMutant [22] and Mahalanobis [50].

With regards to detecting adversarial examples, Table 4 shows that SelfChecker++ achieves much higher accuracies (with an average F1-score of 80.07%) than DISSECTOR (average F1-score of 45.53%). For FPR, even though DISSECTOR achieves lower FPR than SelfChecker++ in several settings, on average, SelfChecker++ with 4.05% is still lower than DISSECTOR with 5.79%. And in such settings where DISSECTOR has lower FPR, its TPR is also very low. These results indicate that SelfChecker++ can always detect more adversarial examples (TP, which represents setting off correct alarms) and overlooks fewer true alarms (FN) than DISSECTOR, indicating the effectiveness of SelfChecker++.

Even though DISSECTOR obtains good AUC results as claimed in [15], its alarm accuracies are low and oscillate when detecting adversarial example generated by different tools. AUC measures how effective a technique is in distinguishing two kinds of data. But with a poor threshold, the technique with higher AUC values will have poor detection accuracy. The significantly different alarm accuracies of DISSECTOR indicate this problem (we selected thresholds for both DISSECTOR and SelfChecker++ from the validation set). DISSECTOR works well when detecting adversarial examples generated by ADAPT and PGSM on FMNIST and works fine for RobOT on MNIST and FMNIST, but it works poorly in other settings. This indicates that DISSECTOR is sensitive to the selection of thresholds, while SelfChecker++ is much robust.

For RQ2, we conclude that SelfChecker++ is effective at detecting adversarial examples with a high F1-scores and that it is tolerant for a selection of thresholds.

5.3.3 RQ3. Advice Accuracy

Depicted in Table 5 is the accuracies on test and adversarial examples for M, M+SC and M+SC++ which represents the original model, original model with SelfChecker advice and original model with SelfChecker++ advice respectively.

The results in Table 5 indicate that SelfChecker++ consistently attains the highest advice accuracies with an average value of 69.57% than M with an average value of 48.68%

TABLE 4
Alarm accuracy on intended adversaries.

Dataset	Metrics	RobOT			ADAPT			PGSM			PGD		
		↑*TPR*	↓*FPR*	↑*F1*	↑*TPR*	↓*FPR*	↑*F1*	↑*TPR*	↓*FPR*	↑*F1*	↑*TPR*	↓*FPR*	↑*F1*
MNIST	DISSECTOR	5.75	0.59	10.82	20.70	0.30	34.22	12.41	0.79	21.89	1.17	0.57	2.30
	SelfChecker++	55.88	0.74	**71.35**	73.48	0.76	**84.36**	64.35	0.93	**77.79**	39.83	0.71	**56.65**
FMNIST	DISSECTOR	53.69	11.76	65.29	76.94	11.05	82.66	63.66	11.21	73.52	55.07	10.98	67.02
	SelfChecker++	77.28	7.31	**84.00**	84.59	7.70	**88.57**	72.31	7.73	**80.85**	66.73	7.65	**77.06**
CIFAR-10	DISSECTOR	45.24	5.07	60.54	33.78	5.85	48.73	36.74	5.80	51.16	17.31	5.55	28.19
	SelfChecker++	76.08	3.75	**84.91**	73.13	3.75	**82.98**	80.92	4.01	**87.16**	76.52	3.56	**85.13**

and SelfChecker with an average value of 51.20%. The advice provided by SelfChecker is generated from the internal layer features without any changes to the original model (retraining, change of model architecture, etc). Therefore, the improvement of the advice accuracy against the original model is not significant. And it uses maximum voting on the KDE inferred classes of selected layers to generate advice for input triggering an alarm. This mechanism is unstable when DNN models are attacked by adversarial examples since layer outputs will be messy. SelfChecker++ uses GAN-based transformation to synthesize new images similar to original ones triggering alarms. And the generated ones retain the general parts as the training data so that more conforming to the the distribution of training data than original ones. Beneficial from this, the perturbations introduced by attackers in the adversarial examples are removed. That's why SelfChecker++ significantly improves the accuracy.

For RQ3, we conclude that SelfChecker++'s advice can significantly improve the accuracy of the original models.

5.3.4 RQ4. GAN-based Transformation

Table 5 shows the accuracies of using GAN-based transformation on all test instances (M+GAN) without combining the GAN with alarm analysis. The results show that the accuracies of M+GAN average 55.27% and decrease substantially as compared to SelfChecker++ (M+SC++) and are sometimes even worse than M. The reason for this is that the original model M is trained to generalize as much as possible to all training data by considering some feature correlations. But, the images generated by the GAN remove features beyond the common training data distribution, losing important feature correlations. Therefore, using GAN-based transformation on all data, especially data on which M is confident, lowers performance. But, when using the GAN on only those instances that trigger an alarm, as SelfChecker++ does, the adversarial perturbations are more likely to be removed and important feature correlations are more likely to remain.

For RQ4, we showed that using the GAN-based transformation on the entire dataset is inadequate. SelfChecker++ combines the alarm analysis with the GAN-based transformation to achieve higher accuracies.

5.3.5 RQ5. Deployment Time

In Table 6, we also compared the time needed by these methods when checking a single inference instance from each DNN classifier. The values in Table 6 for each DNN classifier represent the average times across all datasets mentioned in Table 3 which the classifier is associated with. It can be observed that both SELFORACLE and ConfidNet are the fastest. This is because these two methods utilize two DNN models to compute the outputs. Nonetheless, when it comes to alarm accuracies, these two methods are behind the other methods listed in Table 6. On traditional DNN classifiers, SelfChecker (average of 34.98ms) is faster than DISSECTOR (average of 50.47ms). For SelfChecker++, once an alarm is triggered, the GAN-based transformation is called to synthesize a similar image where a search is need to find proper z^*. The time consumption of SelfChecker++ is the sum of triggering an alarm (close to SelfChecker) and GAN-based transformation (GAN). So, SelfChecker++ costs the longest time where GAN takes the most but it also achieves the highest accuracy.

TABLE 5
Advice accuracy.

Acc	Strategies	RobOT	ADAPT	FGSM	PGD
M	M	50.48	49.28	53.65	53.05
	M+*SC*	56.25	52.93	54.93	54.05
	M+*SC++*	**72.98**	**88.35**	**85.13**	**89.20**
	M+*GAN*	66.50	81.20	76.93	68.88
F	M	47.85	45.48	45.93	45.75
	M+*SC*	51.50	51.93	46.08	45.73
	M+*SC++*	**70.35**	**72.88**	**58.25**	**69.43**
	M+*GAN*	67.90	70.70	56.55	65.63
C	M	45.35	45.30	54.28	47.73
	M+*SC*	47.28	48.00	57.33	48.38
	M+*SC++*	**53.25**	**53.78**	**62.85**	**58.33**
	M+*GAN*	25.15	26.18	28.88	28.75

SC and SC++ stand for SelfChecker and SelfChecker++ respectively, M, F, and C stand for MNIST, FMNIST and CIFAR-10 respectively.

TABLE 6
Deployment time.

Time (ms)	SO	DT	CN	SC	GAN	SC++
ConvNet	0.96	29.74	0.98	26.47	41.58	67.9
VGG-16	1.35	58.34	1.02	35.83	144.44	180.17
ResNet-20	1.79	63.33	1.36	42.63	217.26	258.75
DAVE-2	45.80	-	-	67.78	-	-
Chauffeur	42.66	-	-	63.12	-	-

SO, DT, CN, SC, and SC++ stand for SELFORACLE, DISSECTOR, ConfidNet, SelfChecker, and SelfChecker++, respectively

We think that these timings are considered acceptable over multiple application domains. For example, medical image-based diagnosis or even airport security screening can benefit from SelfChecker++. When considering real-time applications such as self-driving car scenarios, SelfChecker and SELFORACLE needs to do better in terms of latency. It should be noted that when it comes to self-driving cars, time taken to check is naturally high due to the fact that 32 frames are analyzed first before setting off an alarm. Though efficiency is not the objective of this paper, we understand its significance in cyber-physical systems. As such, we intend to parallelize SelfChecker++ to make it more efficient, in particular, using one process to estimate per class density function. This would improve latency by 1/(number of classes).

For RQ5, the deployment time is acceptable on safety-critical applications that care about higher accuracy than time consumption (e.g., processing, recognizing or diagnostics).

RQ6. Layer Selection

Aforementioned in Section 4.2 there was a need to selectively choose the layers which were most applicable in order to improve alarm accuracy and search-based optimization was used to accomplish such a task. Table 7 depicts the results for the VGG-16 and Chauffeur models on FMNIST and self-driving car datasets respectively. The rest of the model and dataset result combinations were omitted due to the fact that they share similar properties. In total, we have three approaches when it comes to layer selection for setting off alarms. We evaluate these three strategies with respect to their accuracies. These strategies are random, full and SC-layer (our own strategy mentioned in Section 4.2) respectively. The random layer strategy, as its name suggests, randomly chooses layers for every class (number of layers selected per class are the same as the third strategy to maintain fairness). The full layer strategy uses the entire layer set. Lastly, the SC-layer strategy uses the validation dataset to select layers. Moreover, to maintain fairness, these three strategies did not utilize the boosting strategy.

TABLE 7
Impact of layer selection on alarm accuracy.

FMNIST	TP	FP	TN	FN	↑ TPR	↓ FPR	↑ F1
Random	280	482	8893	345	44.80	5.14	40.37
Full	209	230	9145	416	33.44	**2.45**	39.29
SC-layer[a]	317	329	9046	308	**50.72**	3.51	**49.88**
Chauffeur	**TP**	**FP**	**TN**	**FN**	**↑ TPR**	**↓ FPR**	**↑ F1**
Random	112	3059	5180	10	91.80	37.13	6.80
Full	99	2596	5643	23	81.15	**31.51**	7.03
SC-layer[a]	116	2978	5261	6	**95.08**	36.15	**7.21**

[a] SC-layer stands for SelfChecker++'s layer selection.

From Table 7, it can be observed that SC-layer attains the best TPR and F1-score against the other strategies. It should be noted that the full strategy attains the best FPR but at the expense of TP (i.e., correct alarms) where it falls short compared to SC-layer (108 for FMNIST and 17 for driving). As such, increasing the amount of layers does not produce a better checker.

For RQ6, we showed that a careful selection of layers allows SelfChecker++ to identify more misclassifications and raise more correct alarms.

RQ7. Boosting Strategy

TABLE 8
Impact of boosting on alarm accuracy checking ResNet-20.

FMNIST	TP	FP	TN	FN	↑ TPR	↓ FPR	↑ F1
SC-*b*	402	323	8951	324	55.37	3.48	55.41
SC	376	91	9183	350	51.79	**0.98**	**63.03**
CIFAR[a]	**TP**	**FP**	**TN**	**FN**	**↑ TPR**	**↓ FPR**	**↑ F1**
SC-*b*	2571	930	6022	477	84.35	13.38	78.52
SC	2468	493	6459	580	80.97	**7.09**	**82.14**

[a] CIFAR stands for CIFAR-100

Table 8 compares the effect of the boosting strategy (mentioned earlier in Section 4.2) on SelfChecker++ in terms of accuracy for ResNet-20 on two datasets. As with before, the rest of the model and dataset result combinations were omitted due to the fact that they share similar properties. It can be observed that the boosting strategy (*SC*) attains better FPR and F1-scores than the strategy without boosting (SC-*b*).

For RQ7, we conclude that the boosting strategy significantly improves alarm accuracy by reducing false alarms.

6 RELATED WORK

Studies that revolve around the trustworthiness of DL models tend to concentrate more on the model engineering aspect: generate adversarial test instances [26], [51], [52], increase test coverage [53], [54], and improve robust accuracy [45]. Moreover, they are all dependent on manually supplied ground truth labels which differs from our work which checks the model during production itself.

6.1 Runtime Trustworthiness Checking

There has been some works in the SE community which considered checking the trustworthiness of a DL model during *deployment*. In particular, Stocco et al. [14] proposed SELFORACLE in order to estimate the confidence of self-driving car models. It is largely based on using the combination of an autoencoder, probability distribution fitting and time series analysis in order to locate the confidence boundary of normal/unsupported conditions. They configured it such that an alarm is set off once the confidence of the model's output goes below a threshold. The caveat is that SELFORACLE was primarily designed for inputs that were temporally ordered, like video frames, and that there were performance limitations in regard to DNN types (see Section 5). DISSECTOR by Wang et al. [15] could detect inputs which deviated from those that were considered normal. It made use of multiple sub-models in addition to the pre-trained DL model for the sole purpose of validating samples that were fed to

it. However, generating sub-models is extremely inefficient and DISSECTOR did not provide a detailed enough schema for distinguishing inputs.

Moving over to the DL community, there are works which propose state-of-the-art learning-based models so as to measure confidence [13], [16], [17]. There are however some caveats here as well as these learning-based models can be untrustworthy and suffer from unwanted problems such as overfitting etc. For example, [16], [17] used nearest-neighbor classifiers in order to gauge a model's confidence. It is needless to say that scalability will always be capped here with the primary bottleneck being the expensive nature of computing nearest neighbors on complex models and large datasets. Another example is ConfidNet as a confidence model, proposed by Corbière et al. [13], is built on top of a pre-trained model and learns the confidence criterion based on True Class Probability to predict failures. When it comes to effectiveness and efficiency, [13] outperforms [17] by a significant margin. [13] however has limitations when it comes to performance due to overfitting since the training dataset it is trained on has few wrong predictions.

With the exception of [16], none of the aforementioned papers suggest an alternative advice. This is where SelfChecker++ surpasses them as it is able to achieve both high alarm and advice accuracy by making use of a DNN's internal features.

6.2 Adversarial Examples Detection

Adversarial samples with small artificial perturbations were generated by adversarial attacking techniques to fool DL models, e.g., FGSM [26], PGD [40], DeepXplore [54], DLFuzz [55], ADAPT [39], and RobOT [38]. FGSM and PGD are traditional adversarial attacks based on the intuition that the prediction of an input sample can be changed by modifying its softmax value to the largest extent based on its gradient. DeepXplore by Pei et al. [54] is a white-box differential testing framework to systematically find inputs that can produce differential behaviors using gradient-based search techniques. DLFuzz proposed by Guo et al. [55] uses differential fuzz testing to maximize neuron coverage by continually minutely mutating the input, and generating more adversarial samples for a given DL system. Lee et al. [39] proposed ADAPT to generate adversarial examples by improving the multi-granularity neuron coverage metrics defined in [56]. RobOT [38] is designed to generate adversarial examples for robustness training by designing new testing metrics based on the first-order loss that quantify the importance of each test case with respect to the model's robustness.

Several studies focus on detecting adversarial examples. Metzen et al. [57] trained a 'detector' sub-model from normal and adversarial samples. Besides, the behavior difference between an adversarial sample and a normal sample in the softmax output is used for discrimination [11], [58]. Recently, Wang et al. [22] proposed mMutant to distinguish between adversarial examples and normal ones based on model mutation analysis with the assumption that the adversarial samples usually lie around classification boundaries. In [15], DISSECTOR is shown to achieve broader generalization and higher AUC values than mMutant. We thus use DISSECTOR as the competitor when answering RQ2 in Section 5.3.

7 Threats to Validity

Our experimental results demonstrate SelfChecker++'s effectiveness. However, we acknowledge some threats to the validity of our approach and experiments. We discuss threats to internal validity, construct validity, and external validity as suggested by Wohlin et al. [59].

For internal validity, SelfChecker++ is a layer-based approach that requires white-box access, leading to more limited power on shallow DNNs with few layers. For construct validity, we used measures like TPR, FPR, F1-score that are also used in [14]. Even though we showed in Section 5.3 that AUC is insufficient to measure the detection ability, other related measurements (e.g., AUPR-Error, AUPR-Success) are interesting to explore in the future.

For external validity to our experimental conclusions, the first one is our selected three DNN models (ConvNet, VGG-16, and ResNet-20) and four datasets (MNIST, FMNIST, CIFAR-10, and CIFAR-100) that we used to evaluate the effectiveness of SelfChecker++. We tried to alleviate this threat as follows: (1) the four datasets have been widely used in prior research [13], [15], [38] and have different topics, labels (from 10 and 100), and scales (60,000 and 70,000 samples); (2) the three DNN models are famous with different model types, number of layers (8-20), and model accuracies (66.79%-99.46%); (3) the inclusion of self-driving datasets and models further generalizes our work beyond existing techniques that only consider traditional DNN classifiers [13], [15] or self-driving car scenarios [14]; (4) we use adversarial examples generated by four tools to evaluate SelfChecker++'s detection accuracy while DISSECTOR only uses one tool. Therefore, our experimental conclusions should generally hold.

The second external validity concern is the selection of competing tools. To mitigate this, we compare SelfChecker++ with SELFORACLE [14], ConfidNet [13], DISSECTOR [15] and SelfChecker [23] that are representative and the latest tools in this space (published during 2019–2021). Besides, we always selected the best variant from SELFORACLE and DISSECTOR as competitors.

The third external validity concern comes from the implementation of DISSECTOR. How to build and train sub-models for different DNN models in DISSECTOR is not published. We implemented this part according to the descriptions presented in [15] and achieved close AUC values for the same settings as [15]. But, we acknowledge that the results may differ and depend on the architectures of sub-models for different DNN models.

8 Conclusion

DNNs are being adopted in a variety of domains. But, when utilized in safety- and security-critical contexts, there is a need to monitor as well as to check their outputs: their probabilistic nature means that DNNs would unavoidably choose the incorrect decision on some inputs (either unintended anomalies or malicious inputs). In this paper we hypothesized that features in internal layers of a DNN

can be used to construct a *self-checking* system to monitor DNN outputs. We detailed the design of a system which addressed this, SelfChecker++. Using SelfChecker++ we conducted experiments to check three neural networks on four datasets. The networks in particular were ConvNet, VGG-16, and ResNet-20. The datasets on the other hand were MNIST, FMNIST, CIFAR-10, and CIFAR-100. For unintended anomalies, SelfChecker++ provides accurate alarms with accuracy of 63.10% and has an F1-score of 67.92%. This is also the best score, 9.15% more than the runner up, ConfidNet. With respect to self-driving cars (i.e., DAVE-2 and Chauffeur), SelfChecker++ was also put to the test against the state-of-the-art technique (i.e., SELFORACLE). It was then observed that SelfChecker++ raised more right alarms and were comparable in wrong alarms. When evaluated on adversarial examples, SelfChecker++ can detect around 70.09% of adversarial examples, which is 34.88% more than DISSECTOR and SelfChecker++-generated advice is able to increase the accuracy of the original model by at most 39.07%.

REFERENCES

[1] M. Y. Tachbelie, A. Abulimiti, S. T. Abate, and T. Schultz, "Dnn-based speech recognition for globalphone languages," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2020, pp. 8269–8273.

[2] J. Wu, Y. Yu, C. Huang, and K. Yu, "Deep multiple instance learning for image classification and auto-annotation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3460–3469.

[3] X. Cheng, L. Zhang, and Y. Zheng, "Deep similarity learning for multimodal medical images," *Computer Methods in Biomechanics and Biomedical Engineering: Imaging & Visualization*, vol. 6, no. 3, pp. 248–252, 2018.

[4] K. Wang, J. Zhang, G. Bai, R. Ko, and J. S. Dong, "It's not just the site, it's the contents: Intra-domain fingerprinting social media websites through cdn bursts," in *Proceedings of the Web Conference 2021*, ser. WWW '21, 2021, p. 2142–2153.

[5] R. Vinayakumar and K. Soman, "Deepmalnet: evaluating shallow and deep networks for static pe malware detection," *ICT express*, vol. 4, no. 4, pp. 255–258, 2018.

[6] J. James, J. J. Ford, and T. L. Molloy, "Below horizon aircraft detection using deep learning for vision-based sense and avoid," in *2019 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, 2019, pp. 965–970.

[7] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang *et al.*, "End to end learning for self-driving cars," *arXiv:1604.07316*, 2016.

[8] S. Mitsch and A. Platzer, "ModelPlex: Verified runtime validation of verified cyber-physical system models," *Formal Methods in System Design*, vol. 49, no. 1-2, pp. 33–74, 2016.

[9] Y. Lin, J. Sun, Y. Xue, Y. Liu, and J. Dong, "Feedback-based debugging," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 393–403.

[10] C. E. Shannon, "A mathematical theory of communication," *Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[11] D. Hendrycks and K. Gimpel, "A baseline for detecting misclassified and out-of-distribution examples in neural networks," *arXiv:1610.02136*, 2016.

[12] V. T. Vasudevan, A. Sethy, and A. R. Ghias, "Towards better confidence estimation for neural models," in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2019, pp. 7335–7339.

[13] C. Corbière, N. Thome, A. Bar-Hen, M. Cord, and P. Pérez, "Addressing failure prediction by learning model confidence," in *Advances in Neural Information Processing Systems*, 2019, pp. 2902–2913.

[14] A. Stocco, M. Weiss, M. Calzana, and P. Tonella, "Misbehaviour prediction for autonomous driving systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 359–371.

[15] H. Wang, J. Xu, C. Xu, X. Ma, and J. Lu, "Dissector: Input validation for deep learning applications by crossing-layer dissection," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 727–738.

[16] N. Papernot and P. McDaniel, "Deep k-nearest neighbors: Towards confident, interpretable and robust deep learning," *arXiv:1803.04765*, 2018.

[17] H. Jiang, B. Kim, M. Guan, and M. Gupta, "To trust or not to trust a classifier," in *Advances in neural information processing systems*, 2018, pp. 5541–5552.

[18] X. Gao, R. K. Saha, M. R. Prasad, and A. Roychoudhury, "Fuzz testing based data augmentation to improve robustness of deep neural networks," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1147–1158.

[19] Z. Zhong, L. Zheng, G. Kang, S. Li, and Y. Yang, "Random erasing data augmentation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 07, 2020, pp. 13 001–13 008.

[20] A. Shafahi, M. Najibi, A. Ghiasi, Z. Xu, J. Dickerson, C. Studer, L. S. Davis, G. Taylor, and T. Goldstein, "Adversarial training for free!" in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, 2019, pp. 3358–3369.

[21] A. Shrivastava, T. Pfister, O. Tuzel, J. Susskind, W. Wang, and R. Webb, "Learning from simulated and unsupervised images through adversarial training," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 2107–2116.

[22] J. Wang, G. Dong, J. Sun, X. Wang, and P. Zhang, "Adversarial sample detection for deep neural network through model mutation testing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1245–1256.

[23] Y. Xiao, I. Beschastnikh, D. S. Rosenblum, C. Sun, S. Elbaum, Y. Lin, and J. S. Dong, "Self-checking deep neural networks in deployment," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 372–384.

[24] M. H. Meng, G. Bai, S. G. Teo, Z. Hou, Y. Xiao, Y. Lin, and J. S. Dong, "Adversarial robustness of deep neural networks: A survey from a formal verification perspective," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[25] C. Laidlaw and S. Feizi, "Functional adversarial attacks," *Advances in neural information processing systems*, vol. 32, 2019.

[26] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[27] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[28] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio, "Generative adversarial nets," in *NIPS*, 2014.

[29] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," 2016.

[30] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," 2016.

[31] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, "Gans trained by a two time-scale update rule converge to a local nash equilibrium," 2018.

[32] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "Testing deep neural networks," *arXiv:1803.04792*, 2018.

[33] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-CAM: Visual explanations from deep networks via gradient-based localization," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 618–626.

[34] Y. Liang, K. Ouyang, L. Jing, S. Ruan, Y. Liu, J. Zhang, D. S. Rosenblum, and Y. Zheng, "UrbanFM: Inferring fine-grained urban flows," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2019, pp. 3132–3142.

[35] R. A. Davis, K.-S. Lii, and D. N. Politis, "Remarks on some nonparametric estimates of a density function," in *Selected Works of Murray Rosenblatt*. Springer, 2011, pp. 95–100.

[36] L. Cohen, P. Jarvis, and J. Fowler, *Practical statistics for field biology*. John Wiley & Sons, 2013.

[37] P. Samangouei, M. Kabkab, and R. Chellappa, "Defense-gan: Protecting classifiers against adversarial attacks using generative models," *arXiv preprint arXiv:1805.06605*, 2018.

[38] J. Wang, J. Chen, Y. Sun, X. Ma, D. Wang, J. Sun, and P. Cheng, "Robot: Robustness-oriented testing for deep learning systems,"

in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 300–311.
[39] S. Lee, S. Cha, D. Lee, and H. Oh, "Effective white-box testing of deep neural networks with adaptive neuron-selection strategy," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 165–176.
[40] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.
[41] Y. Kaya, S. Hong, and T. Dumitras, "Shallow-deep networks: Understanding and mitigating network overthinking," in *International Conference on Machine Learning*, 2019, pp. 3301–3310.
[42] Y. LeCun, C. Cortes, and C. Burges, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/
[43] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.
[44] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html
[45] J. Kim, R. Feldt, and S. Yoo, "Guiding deep learning system testing using surprise adequacy," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1039–1049.
[46] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv:1409.1556*, 2014.
[47] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
[48] T. Chauffeur. (2019) Steering angle model: Chauffeur. [Online]. Available: https://github.com/udacity/self-driving-car/tree/master/steering-models/community-models/chauffeur
[49] D. W. Scott, *Multivariate density estimation: theory, practice, and visualization*. John Wiley & Sons, 2015.
[50] K. Lee, K. Lee, H. Lee, and J. Shin, "A simple unified framework for detecting out-of-distribution samples and adversarial attacks," *Advances in neural information processing systems*, vol. 31, 2018.
[51] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "DeepFool: a simple and accurate method to fool deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2574–2582.
[52] Q. Li, Y. Qi, Q. Hu, S. Qi, Y. Lin, and J. S. Dong, "Adversarial adaptive neighborhood with feature importance-aware convex interpolation," *IEEE Transactions on Information Forensics and Security*, 2020.
[53] Y. Tian, K. Pei, S. Jana, and B. Ray, "DeepTest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the International Conference on Software Engineering*, 2018, pp. 303–314.
[54] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated whitebox testing of deep learning systems," in *proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.
[55] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "Dlfuzz: Differential fuzzing testing of deep learning systems," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 739–743.
[56] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu *et al.*, "Deepgauge: Multi-granularity testing criteria for deep learning systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 120–131.
[57] J. H. Metzen, T. Genewein, V. Fischer, and B. Bischoff, "On detecting adversarial perturbations," *arXiv preprint arXiv:1702.04267*, 2017.
[58] S. Liang, Y. Li, and R. Srikant, "Enhancing the reliability of out-of-distribution image detection in neural networks," in *International Conference on Learning Representations*, 2018.
[59] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

Yan Xiao is a Research Fellow at National University of Singapore. She received her PhD degree from the City University of Hong Kong in 2019. Her research focuses on trustworthiness of deep learning system and AI applications in software engineering. More information is available on her homepage: https://yanxiao6.github.io/.

Ivan Beschastnikh is an associate professor in the Department of Computer Science at the University of British Columbia. He received his PhD from the University of Washington in 2013. He has broad research interests that touch on systems and software engineering. His recent projects span distributed systems, program analysis, and machine learning. More information is available on his homepage: http://www.cs.ubc.ca/ bestchai/.

Yun Lin received his PhD from Fudan University. He is currently a Research Assistant Professor in National University of Singapore. His research mainly focuses on program analysis, traditional software testing and quality assurance analysis of artificial intelligence. He has published some top tier conference/journal papers relevant to software analysis in ICSE, ISSTA, FSE, ASE, TIFS, TSE, AAAI, and USENIX Security. In particular, he won ACM SIGSOFT Distinguished Paper Awards in ICSE'18.

Rajdeep Singh Hundal is currently a Research Assistant at National University of Singapore. He received his Masters degree from National University of Singapore in 2022. His research focuses on deep learning systems.

Xiaofei Xie received his Ph.D, M.E. and B.E. from Tianjin University. He is currently an assistant professor in Singapore Management University, Singapore. His research mainly focuses on program analysis, traditional software testing and quality assurance analysis of deep learning systems. He has published some top tier conference/journal papers relevant to software analysis in ICSE, ISSTA, FSE, ASE, TDSC, TIFS, TSE, IJCAI, NeurIPS, ICML and CCS. In particular, he won two ACM SIGSOFT Distinguished Paper Awards in FSE'16 and ASE'19.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3200421

David S. Rosenblum (Fellow, IEEE) is Planning Research Corporation Professor and Chair of the Department of Computer Science at George Mason University. He received his PhD degree from Stanford University and has held positions at AT&T Bell Labs (Murray Hill), University of California, Irvine, PreCache (a technology startup funded by Sony Music), University College London and National University of Singapore. His research interests involve problems in formal methods, software engineering, distributed systems, ubiquitous computing and machine learning, and he has received two test-of-time awards for his research papers. He is a fellow of the ACM and received the 2018 ACM SIGSOFT Distinguished Service Award.

Jin Song Dong received his PhD degree from the University of Queensland, Australia. He is a professor with the School of Computing, National University of Singapore. His research interests include software engineering, program analysis, formal verification, and model checking.